

深度学习

21天实战Caffe

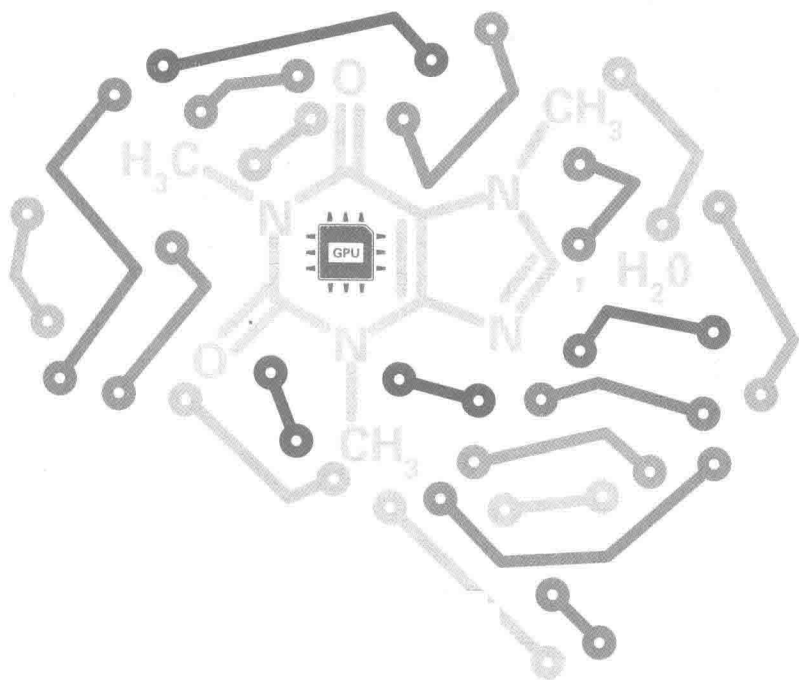
赵永科 著



www.aibbt.com 让未来触手可及
中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



深度学习

21天实战Caffe

赵永科 著

电子工业出版社
www.eipbt.com 让未来触手可及
Publishing House of Electronics Industry
北京·BEIJING

内 容 提 要

本书是一本深度学习入门读物。以目前已经大量用于线上系统的深度学习框架 Caffe 为例,由浅入深,从 Caffe 的配置、部署、使用开始学习,通过阅读 Caffe 源码理解其精髓,加强对深度学习理论的理解,最终达到熟练运用 Caffe 解决实际问题的目的。和国外机器学习、深度学习大部头著作相比,本书偏重动手实践,将难以捉摸的枯燥理论用浅显易懂的形式表达出来,透过代码揭开其神秘面纱,更多地贴近实际应用。

本书非常适合:对人工智能、机器学习感兴趣的读者;希望用深度学习完成设计的计算机或电子信息专业学生;准备开设机器学习、深度学习实践课的授课老师;学习过 C++,希望进一步提升编程水平的开发者;刚入坑的机器学习、语音、机器视觉、智能机器人研发或算法工程师。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

深度学习:21天实战 Caffe / 赵永科著. —北京:电子工业出版社,2016.7

ISBN 978-7-121-29115-9

I. ①深… II. ①赵… III. ①学习系统 IV. ①TP273

中国版本图书馆 CIP 数据核字(2016)第 137508 号

策划编辑:张春雨

责任编辑:葛娜

印 刷:三河市双峰印刷装订有限公司

装 订:三河市双峰印刷装订有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编:100036

开 本:787×980 1/16 印张:24.5 字数:530 千字

版 次:2016 年 7 月第 1 版

印 次:2016 年 7 月第 2 次印刷

定 价:79.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zltts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:(010) 51260888-819, faq@phei.com.cn。

推荐序一

近年来，几乎整个智能学科的研究者们都注意到一个技术名词——深度学习（Deep Learning）。这个略带神秘色彩的名字和其代表的前沿性技术被著名的《MIT 技术评论》评选为 2013 年世界 10 大突破性技术之首。而在此之前，包括 Google、Microsoft、Facebook 等公司在内的诸多信息科技巨头都已争相在此技术上投入了前所未有的重视力度和战略资源，继而高调宣布布局智能应用领域。学术界和工业界不遗余力地抢占相关研究和技术的制高点，人们并没有感到奇怪，因为所有人都明白：这也许是人类在探索人工智能的伟大旅程和漫漫征途上的重要一刻。

关于神经网络的研究可以追溯到 20 世纪 40 年代。在其漫长的历史上经历了数次戏剧性的波折。然而近年来，随着大量数据的获得、先进理论的发现，以及高性能并行计算技术的发展，以深度神经网络为载体的特征学习技术相继在语音、视觉、语言等诸多研究领域中取得了突破性的成果，并且正以不可阻挡之势“入侵”传统技术占领的各个领域。

随着深度学习技术在学术界和工业界得到广泛认可，越来越多的人开始参与到深度学习的相关研究和实践中来。然而，由于存在一定的技术门槛，快速入手深度学习的研究并不是一件容易的事情。其中的一个重要原因是，深度学习中的许多问题非常依赖于实践。然而长期以来，学术界和工业界缺少一款专门为深度学习而设计的，兼具性能、灵活性和扩展性等诸多优势于一身的开源框架。这使得无论是快速实现算法，还是复现他人的结论，都存在着实践上的困难。研究人员和工程师们迫切需要一套通用而高效的深度学习开源框架。

2013 年，一款叫作“Caffe”的深度学习框架由加州大学伯克利分校的@贾扬清博士在 Github 上发布。发布伊始，Caffe 框架就得到了广泛的关注。Caffe 框架以“层”为单位对神经网络的结构进行了高度的抽象，通过一些精巧的设计显著优化了执行效率，并且在保持高效实现的基础上不失灵活性。无论在结构、性能上，还是代码质量上，Caffe 都是一款十分出色的开源框架。更重要的是，它将深度学习的每一个细节都原原本本地展现出来，供人们学习和实践。可以说，Caffe 框架的发布极大地降低了深度学习研究和开发的难度。

正是由于上述的诸多优势，Caffe 框架迅速流行起来，并且逐步形成了强大的用户社区。经过两年多的版本迭代，Caffe 框架已经在学术界和工业界得到了广泛的认可。在学术界，目前每天都有以 Caffe 框架作为底层实现的研究成果发布；而在工业界，已经有许多产品使用 Caffe 作为其深度学习算法实现的内核。从学术界到工业界，大家可以共享同一套底层代码，基于同一

套平台进行研究、交流和生产，这是一件令人愉悦的事情。可以说，Caffe 开源项目对于促进整个深度学习研究领域的快速发展具有不可磨灭的贡献。

对于刚刚接触深度学习的朋友们来说，通过结合 Caffe 的代码来加深对理论的理解，也许是一种事半功倍的方法。Caffe 框架天然的清晰层次和良好的代码可读性，为入手深度学习的朋友们提供了教科书般的实践指导。然而，由于 Caffe 中有大量技术细节是在论文中无法找到的，想要快速理解 Caffe 框架内部的种种精要往往需要费一番周折。幸好，有一些先行者为大家分享了相关的知识。

几天前，我有幸接到好友@卜居的邀请，为他的《深度学习：21 天实战 Caffe》新书做序。这本书是国内第一本在代码级别上全面剖析 Caffe 框架的指导书，同时也是一本真正的实战手册。本书涉及深度学习的基本理论、Caffe 的设计思想、Caffe 中各模块的具体实现，以及各种实例等内容。书中对 Caffe 框架的分析非常细致，涵盖的内容也颇为丰富，可以说是一本入手 Caffe 实践的技术手册，因此特别适合于 Caffe 的初学者阅读。相信本书可以帮助朋友们少走许多弯路。有关 Caffe 的诸多奥秘，@卜居将会在书中为您一一呈现。

感谢创立和推动深度学习研究的科学家们，感谢 Caffe 框架的作者贾扬清博士，感谢本书的作者@卜居，以及所有为深度学习技术的发展而奋斗的朋友们。我们的征途是星辰与大海，让我们一起努力，向着实现人工智能的伟大目标前进！

辛淼

Caffe 中国用户社区 (caffecn.cn) 创始人

推荐序二

在形形色色的科技创新报道中，人工智能（AI）成果引发的社会性冲击无疑前所未有的。一些科技术语如“深度学习”，对信息技术类学生而言变得不再陌生。虽然国际科技界对相关领域的研究已有数十年的历史，而跳跃式进步还只是最近若干年的事。正是这个原因，关于机器智能和“深度学习”的学习类书籍大多偏重理论，或散见于外刊上发表的研究论文、各个研究发展机构的研究报告、开源资料等，鲜有从工程实践出发系统地介绍深度学习的书籍。

国外研究机构设置的开源社区的繁荣发展，从工程实现方面补充了理论研究的不足。然而，对开源代码的阅读、理解、应用对于非机器学习专业人士有较大的挑战性。阿里云计算有限公司的赵永科工程师（博客昵称：卜居）在研发实践的基础上，对深度学习从基础理论到编程实践进行了系统的整理，形成了《深度学习：21 天实战 Caffe》一书。这是一个有技术深度、处于国际技术竞争中的领域；而本书是一个研发亲历者对技术深入理解后的总结，十分难得。

本书的写作风格是引导性的，围绕深度学习基础，通过代码导读方式，循序渐进，揭开了深度学习的神秘面纱，让深度学习技术，包括理论和工程实现，贴近所有 AI 爱好者。相信本书的出版能够激发更多研究者的兴趣，推动 AI 技术在中国的发展和应用。

邹谋炎

中国科学院大学教授，研究员，洪堡学者

推荐序三

让机器具有人类的智能是科学家们从计算机诞生开始就一直在努力的方向，但是由于选择了基于规则的算法，效果一直得不到大的提升，论文中经常以效果比乱猜好作为结论。卷积神经网络的发明者 Geoffrey Hinton 在 20 世纪 70 年代就已经提出了今天的深度学习理论，但是限于当时的计算能力，一直不被人重视。21 世纪以来，随着 NVIDIA GPU 的广泛应用，人工神经网络发挥了它应有的价值，成为今天人工智能的代表性成就，Hinton 也被尊称为鼻祖。

当 GPU 在深度学习领域大范围使用时，研究人员遇到了一个重大的问题——要写大量的复杂的神经网络代码，这带来了巨大的困难。在这个历史性的关键时刻，贾扬清同学开发的 Caffe 适时地出现了，Caffe 让只要会 C++ 编程的人员就可以编写深度学习代码，一下子就降低了深度学习的门槛。随后 Caffe 得到了广泛使用，并且获得了社区的广泛支持，也得到了 NVIDIA 的大力支持，获得了充足的发展，几乎可以说不知道 Caffe 就不能说会深度学习。

Caffe 把深度学习的门槛降低了很多，但是实际上依旧需要了解大量的代码细节才能对其进行修改，而深度学习又是一个计算密集的应用，如何写出高效的代码也非常重要。卜居做过许多有关 Caffe 的工作，包括优化卷积算法，非常了解 Caffe 框架的各个细节，他编写的《深度学习：21 天实战 Caffe》一书非常详细、专业。

卜居用人类的恋爱过程来比喻深度学习的学习过程，从初识、热恋到升华，很让人称道。在初识阶段，从深度学习的概念、历史开始，介绍深度学习和基本理论与传统机器学习算法的不同，也包含了业界对深度学习的反思。在热恋阶段，在具体实操方面，从 Caffe 的安装开始介绍，到具体运行 mnist 数据集；从 Caffe 的目录结构、不同层（功能）和数据抽象的实现细节，到如何求解一个深度学习模型，卜居都一一精确地解读。在升华阶段，卜居详细地解说了 Caffe 支持的 NVIDIA GPU 加速工具 CUDA 和 cuDNN，然后介绍了 Caffe 可视化方法，以及如何在生产环境中部署训练好的 Caffe 模型。

本书虽然以 21 天起名，但是其真实内容是需要读者每天 24 小时学习才能够完全掌握的，相信读者会一天 24 小时手不释卷。

我郑重地向大家推荐此书。

风辰

并行计算领域专家，深度学习平台架构师

前言

缘起

2014 年，我开始在阿里云进行深度学习平台优化，使用了开源框架 Caffe，其间在多种 GPU 服务器上部署运行，同时也根据内部业务需求修改源码，研究基于 FFT 的卷积层加速方法（有幸与 Facebook AI Research Yann LeCun 老爷子同时做同一件事）。在学习和修改 Caffe 源码过程中根据个人理解写了数篇博客，本是无心插柳，孰料有一天博文视点编辑@永恒的侠少找到我，建议将博客内容进一步扩展为一本深度学习入门书，贴近实际，让更多读者走近这个如火如荼的领域。理想远大，现实残酷，互联网公司的工作强度大，常常加班，还要抽时间梳理写作思路，迟迟不能交出一份满意的书稿。熬夜艰辛，码字劳苦，时而思如泉涌，时而困顿踌躇，磨蹭半年有余，总算付梓。

在写作过程中，从头到尾重新审视了一遍近年大火的深度学习技术，提炼了经典论文的精华，深深为其近 30 年的坎坷历程唏嘘不已。我们目前看到的深度学习模型和基本理论可能比我们的年龄都要大，受限于当时计算能力不足，数据集也相对匮乏，深度学习一度陷入研究低谷，直到云计算、大数据时代的到来，日益增长的数据和计算能力为深度学习提供了适宜的温度和土壤，也为其突飞猛进的成长提供了充足的养料。

我们赶上了好时代。在物质和文化如此繁荣之时，有更多的人愿意思考未来——一个充满人工智能技术的时代。自动驾驶汽车、智能机器人、无人机……很多科幻电影中的技术正在成为现实，这一切都得益于深度学习技术和相应软硬件系统的发展进步。通过本书内容，读者将逐步走进深度学习，了解其过去、现在和未来。

本书读者

本书非常适合以下读者：

- 对人工智能、机器学习感兴趣的读者；
- 希望用深度学习完成设计的计算机或电子信息专业学生；

- 准备开设机器学习、深度学习实践课的授课老师；
- 学习过 C++，希望进一步提升编程水平的开发者；
- 刚入坑的机器学习、语音、机器视觉、智能机器人研发或算法工程师。

致谢

首先要感谢@永恒的侠少从茫茫人海中找到了我，出版深度学习入门书籍的想法我们一拍即合。然而，真正执笔才发现困难重重，业余时间常常被加班挤占，进度一拖再拖，在侠少不厌其烦一而再再而三地敦促下才将毫无条理的博客风格文章整理成书，自己也从中重新认识了深度学习，克服了拖延症，战胜了懒惰。

特别感谢我的爱人@晓曼在我情绪低落无心写作时给予的悉心照顾和支持，一次次谈心让我转换新的思路，跳出局部极小值点，坚定迈向下一个目标。

感谢远在家乡的父母，他们年过花甲，不懂我写的“深度学习”是什么，但他们知道我忙，知道我在为了理想付出，每次打电话都记挂着我，一次次叮嘱少熬夜。掩卷深思，自己欠父母太多，只愿能抽出更多的时间陪陪他们。

在阿里云工作已有近两年时间，非常 Nice 的同事们也对我的成长起到至关重要的作用，感谢老大@长仁提供良好的学习平台和环境，让我有机会接触最新架构 CPU、GPU、Phi、FPGA，通过一次次折腾机器，不断打怪升级攒经验。愿 2016 年我们的阿里云高性能计算服务 (<https://www.aliyun.com/product/hpc>) 随深度学习技术一起牛 X 一起飞！

感谢 CafféCN 社区 (<http://www.caffecn.cn/>) 的@辛淼博士创建国内最前沿的深度学习交流平台，在这里聚集了一批始终代表先进生产力的 CS 硕博研究生、企业一线工程师，通过学术和技术讲座，让不同的思维火花碰撞，触发无限可能，欢迎更多的有志之士加入！

感恩互联网时代，感恩深度学习领域的先驱者 Geoffrey Hinton、Yann LeCun、YoushuaBengio，感恩青年才俊 Alex Krizhesky、YangqingJia 将生冷的深度学习理论转化为开源代码吸引了大量的热血青年前赴后继投身该领域，共建社区。

赵永科（笔名：卜居）

2016 年 4 月于杭州蓝湾 Coffee

CSDN 博客地址：<http://blog.csdn.net/kkk584520>

电子邮箱：zhaoyongke@yeah.net

目 录

上篇 初见

第 1 天	什么是深度学习	2
1.1	星星之火，可以燎原	3
1.2	师夷长技	4
1.2.1	谷歌与微软	4
1.2.2	Facebook、亚马逊与 NVIDIA	5
1.3	中国崛起	6
1.3.1	BAT 在路上	6
1.3.2	星光闪耀	7
1.3.3	企业热是风向标	8
1.4	练习题	9
第 2 天	深度学习的过往	10
2.1	传统机器学习的局限性	10
2.2	从表示学习到深度学习	11
2.3	监督学习	12
2.4	反向传播算法	13
2.5	卷积神经网络	15
2.6	深度学习反思	17
2.7	练习题	18
2.8	参考资料	18
第 3 天	深度学习工具汇总	19
3.1	Caffe	19
3.2	Torch & OverFeat	20
3.3	MxNet	22
3.4	TensorFlow	22
3.5	Theano	24

3.6	CNTK	24
3.7	练习题	25
3.8	参考资料	26
第 4 天	准备 Caffe 环境	27
4.1	Mac OS 环境准备	27
4.2	Ubuntu 环境准备	28
4.3	RHEL/Fedora/CentOS 环境准备	29
4.4	Windows 环境准备	29
4.5	常见问题	32
4.6	练习题	32
4.7	参考资料	33
第 5 天	Caffe 依赖包解析	34
5.1	ProtoBuffer	34
5.2	Boost	38
5.3	GFLAGS	38
5.4	GLOG	39
5.5	BLAS	40
5.6	HDF5	41
5.7	OpenCV	42
5.8	LMDB 和 LEVELDB	42
5.9	Snappy	43
5.10	小结	43
5.11	练习题	49
5.12	参考资料	49
第 6 天	运行手写体数字识别例程	50
6.1	MNIST 数据集	50
6.1.1	下载 MNIST 数据集	50
6.1.2	MNIST 数据格式描述	51
6.1.3	转换格式	53
6.2	LeNet-5 模型	60
6.2.1	LeNet-5 模型描述	60

6.2.2	训练超参数	65
6.2.3	训练日志	66
6.2.4	用训练好的模型对数据进行预测	76
6.2.5	Windows 下训练模型	76
6.3	回顾	78
6.4	练习题	79
6.5	参考资料	79
篇尾语		80

中篇 热恋

第 7 天	Caffe 代码梳理	82
7.1	Caffe 目录结构	82
7.2	如何有效阅读 Caffe 源码	84
7.3	Caffe 支持哪些深度学习特性	86
7.3.1	卷积层	86
7.3.2	全连接层	89
7.3.3	激活函数	91
7.4	小结	99
7.5	练习题	99
7.6	参考资料	100
第 8 天	Caffe 数据结构	101
8.1	Blob	101
8.1.1	Blob 基本用法	102
8.1.2	数据结构描述	108
8.1.3	Blob 是怎样炼成的	109
8.2	Layer	125
8.2.1	数据结构描述	126
8.2.2	Layer 是怎样建成的	127
8.3	Net	136
8.3.1	Net 基本用法	136
8.3.2	数据结构描述	139
8.3.3	Net 是怎样绘成的	139

8.4	机制和策略	146
8.5	练习题	147
8.6	参考资料	148
第 9 天	Caffe I/O 模块	149
9.1	数据读取层	149
9.1.1	数据结构描述	149
9.1.2	数据读取层实现	150
9.2	数据变换器	155
9.2.1	数据结构描述	155
9.2.2	数据变换器的实现	156
9.3	练习题	171
第 10 天	Caffe 模型	172
10.1	prototxt 表示	173
10.2	内存中的表示	176
10.3	磁盘上的表示	176
10.4	Caffe Model Zoo	178
10.5	练习题	180
10.6	参考资料	180
第 11 天	Caffe 前向传播计算	181
11.1	前向传播的特点	181
11.2	前向传播的实现	182
11.2.1	DAG 构造过程	182
11.2.2	Net Forward 实现	190
11.3	练习题	192
第 12 天	Caffe 反向传播计算	193
12.1	反向传播的特点	193
12.2	损失函数	193
12.2.1	算法描述	194
12.2.2	参数描述	195
12.2.3	源码分析	195

12.3 反向传播的实现	203
12.4 练习题	205
第 13 天 Caffe 最优化求解过程	207
13.1 求解器是什么	207
13.2 求解器是如何实现的	208
13.2.1 算法描述	208
13.2.2 数据结构描述	210
13.2.3 CNN 训练过程	218
13.2.4 CNN 预测过程	225
13.2.5 Solver 的快照和恢复功能	227
13.3 练习题	230
第 14 天 Caffe 实用工具	231
14.1 训练和预测	231
14.2 特征提取	241
14.3 转换图像格式	247
14.4 计算图像均值	254
14.5 自己编写工具	257
14.6 练习题	257
篇尾语	258

下篇 升华

第 15 天 Caffe 计算加速	260
15.1 Caffe 计时功能	260
15.2 Caffe GPU 加速模式	262
15.2.1 GPU 是什么	262
15.2.2 CUDA 是什么	263
15.2.3 GPU、CUDA 和深度学习	263
15.2.4 Caffe GPU 环境准备	264
15.2.5 切换到 Caffe GPU 加速模式	268
15.3 Caffe cuDNN 加速模式	269
15.3.1 获取 cuDNN	270

15.3.2	切换到 Caffe cuDNN 加速模式	270
15.3.3	Caffe 不同硬件配置性能	272
15.4	练习题	273
15.5	参考资料	273
第 16 天	Caffe 可视化方法	275
16.1	数据可视化	275
16.1.1	MNIST 数据可视化	275
16.1.2	CIFAR10 数据可视化	277
16.1.3	ImageNet 数据可视化	278
16.2	模型可视化	279
16.2.1	网络结构可视化	279
16.2.2	网络权值可视化	281
16.3	特征图可视化	288
16.4	学习曲线	295
16.5	小结	298
16.6	练习题	298
16.7	参考资料	299
第 17 天	Caffe 迁移和部署	300
17.1	从开发测试到生产部署	300
17.2	使用 Docker	302
17.2.1	Docker 基本概念	302
17.2.2	Docker 安装	303
17.2.3	Docker 入门	305
17.2.4	Docker 使用进阶	312
17.3	练习题	317
17.4	参考资料	317
第 18 天	关于 ILSVRC 不得不说的一些事儿	318
18.1	ImageNet 数据集	318
18.2	ILSVRC 比赛项目	319
18.2.1	图像分类 (CLS)	320
18.2.2	目标定位 (LOC)	320

18.2.3	目标检测 (DET)	321
18.2.4	视频目标检测 (VID)	322
18.2.5	场景分类	322
18.3	Caffe ILSVRC 实践	323
18.4	练习题	326
18.5	参考资料	326
第 19 天	放之四海而皆准	327
19.1	图像分类	327
19.1.1	问题描述	327
19.1.2	应用案例——商品分类	330
19.2	图像中的字符识别	332
19.2.1	问题描述	332
19.2.2	应用案例——身份证实名认证	333
19.3	目标检测	337
19.3.1	问题描述	337
19.3.2	最佳实践——运行 R-CNN 例程	337
19.4	人脸识别	340
19.4.1	问题描述	340
19.4.2	最佳实践——使用 Face++ SDK 实现人脸检测	342
19.5	自然语言处理	343
19.5.1	问题描述	343
19.5.2	最佳实践——NLP-Caffe	344
19.6	艺术风格	350
19.6.1	问题描述	350
19.6.2	最佳实践——style-transfer	352
19.7	小结	354
19.8	练习题	354
19.9	参考资料	355
第 20 天	继往开来的领路人	356
20.1	Caffe Traps and Pitfalls	356
20.1.1	不支持任意数据类型	356
20.1.2	不够灵活的高级接口	357

20.1.3	繁杂的依赖包	357
20.1.4	堪忧的卷积层实现	357
20.1.5	架构之殇	358
20.1.6	应用场景局限性	358
20.2	最佳实践——Caffe2	359
20.3	练习题	361
20.4	参考资料	362
第 21 天	新生	363
21.1	三人行，必有我师	363
21.2	路漫漫其修远兮，吾将上下而求索	364
篇尾语	366
结束语	367
附录 A	其他深度学习工具	368

上篇 初见

人生若只如初见，何事秋风悲画扇。

等闲变却故人心，却道故人心易变。

骊山语罢清宵半，泪雨零铃终不怨。

何如薄幸锦衣郎，比翼连枝当日愿。

——纳兰性德《木兰词·拟古决绝词柬友》

第一次见到 Caffe，是 2014 年年初我在阿里云实习期间，看到很多同事在用 Caffe 做深度学习算法优化。当时被名字吸引了，因为我之前只知道 Java 是一种咖啡，而这个深度学习框架全称是“快速特征植入的卷积结构”（Convolutional Architecture for Fast Feature Embedding）。当时的 Caffe 还只是雏形，看了一遍代码后深深被其设计所吸引，高效的 C++/CUDA 实现、Matlab/Python 接口、独特的网络描述方式、清晰的代码框架……当时对深度学习的认识仅仅停留在教科书中的“神经网络”、“多层感知器”、“径向基函数”等抽象概念上，没有具体可操作性，而读完 Caffe 代码就像突然发现了新大陆一般，原来深度学习实现起来居然可以这么简单！

经过两年的发展，Caffe 已经发生了巨大的变化，例如加速库 cuDNN 引入、多 GPU 支持、多种计算平台（AWS、Apache Spark、阿里云 ODPS/HPC）支持等。其核心变化并不大，体现了其设计模式的前瞻性。Caffe 文档丰富，对初学者的友好程度大大提升。本篇的目的是让 Caffe 走近每个深度学习爱好者，亲身体验深度学习最佳实践方式。

第1天

什么是深度学习

第一眼看到“学习”，大多数人想到的是读书、上课、写作业，我们就拿它作为切入点。上课时，我们是跟着老师一步步学习，即“有监督”学习；而课后的作业，则需要靠自己完成，是“无监督”学习。平时做的课后练习题，是我们学习系统的“训练数据集”，而考试时卷面上的题目则属于“测试数据集”，用于检验我们的学习成果。“学霸”训练效果比其他人好，对测试数据集的所有情况如数家珍；“学渣”则完全没有训练或训练不充分，对测试数据集的效果和随机猜测差不多；还有“学痴”在训练上出现了“过拟合”，平时做训练题滚瓜烂熟，一遇大考就跪了……

更抽象地表达，可以说学习是一个不断发现自身错误并改正错误的迭代过程。人是如此，机器亦如此。带“学习”功能的机器能仅仅通过“看”未知系统的输入-输出对（称为**训练样本**），自动实现该系统内部算法，并具有举一反三的能力（称为**泛化**），对不在训练样本中的未知输入也能产生正确的输出，完全不需要程序员或算法专家动手设计中间算法，是不是感觉非常酷？

如果将训练样本表示为：

$$D = \{z_1, z_2, \dots, z_n\}$$

其中， z_i 表示未知系统 $P(Z)$ 中采样得到的数据（每个元素都可表示为输入-输出对）。规定惩罚函数 $L(f, Z)$ ，其参数为学习到的规则 f 和独立于训练样本的验证样本集 Z ，其返回值为实数标量，称为惩罚值，又称损失（Loss）。对机器的要求是让损失最小，否则会让机器陷入无止境的重复计算中不得安宁。运行在这些机器上的邪恶算法称为机器学习算法，它能从数据 D 中学习游戏规则 f （攒经验），然后靠学到的经验不断提高游戏得分，最终获得整套游戏攻略（训练好的模型）。

为了让机器自动学习，需要为机器准备三份数据：

(1) 训练集，机器学习的样例。

(2) 验证集，机器学习阶段，用于评估得分和损失是否达到预期要求。

(3) 测试集，机器学习结束之后，实战阶段评估得分。

在现代社会，机器学习技术增强了网页搜索、社交网络内容过滤、电子商务网站广告推荐系统、消费产品如相机和智能手机等方面的能力。

机器学习系统用于识别图像中的物体、将语音转换为文字、匹配新闻条目、在搜索中选择相关度更高的内容，以及推荐用户感兴趣的商品等。在这些应用中越来越多地使用一类叫作深度学习（Deep Learning, DL）的技术。深度学习是由多个处理层组成的计算模型，可以通过学习获得数据的多抽象层表示。该方法显著提高了语音识别、视觉目标识别和检测效果，很多领域（如药品发明、基因测序）也从中受益。

国内外大量互联网公司、高校和科研机构逐步加大深度学习技术的投入，在越来越多的实际项目中加以应用（见图 1-1）。

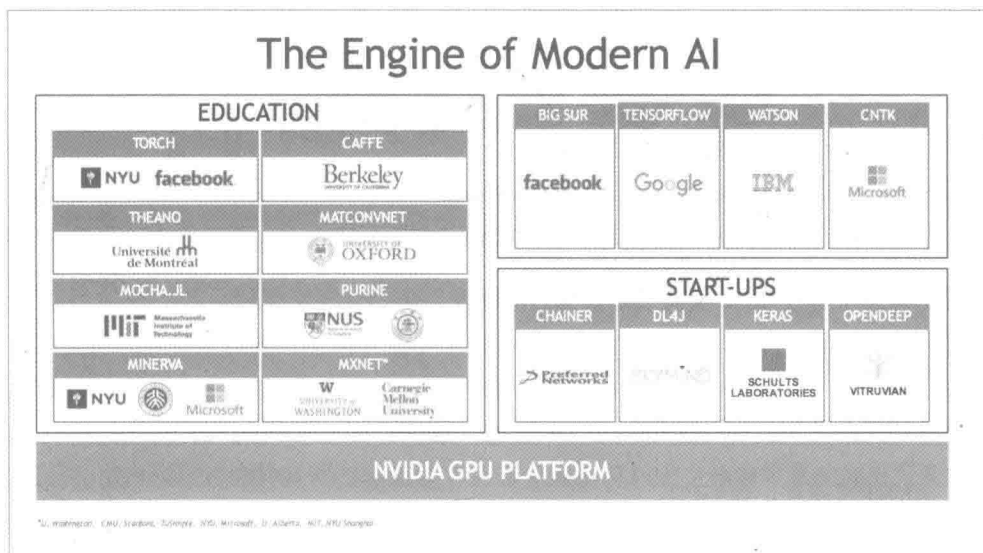


图1-1 国外使用深度学习技术的机构

1.1 星星之火，可以燎原

深度学习一度备受冷落，真正的燎原之势始于 2012 年多伦多大学 Geoffrey Hinton 的学生 Alex Krizhesky 在 ILSVRC (ImageNet Large Scale Visual Recognition Challenge, ImageNet 大规

模视觉识别竞赛, <http://image-net.org/challenges/LSVRC/>) 中使用深度学习方法一举夺得图像分类、目标定位两个项目冠军, 远远拉开了与第二名 (传统计算机视觉方法) 成绩的差距。

如图 1-2 所示为 Alex 在比赛中使用的深度学习模型 AlexNet 结构。注意到网络分成上下两部分, 分别运行在两块 GPU (Graphics Processing Unit) 上, 其中虚线表示两块 GPU 之间的数据通信。事实上, 该模型已成为深度学习的模板结构, 一些新模型 (VGG/GoogLeNet) 均在 AlexNet 基础上改进得到。

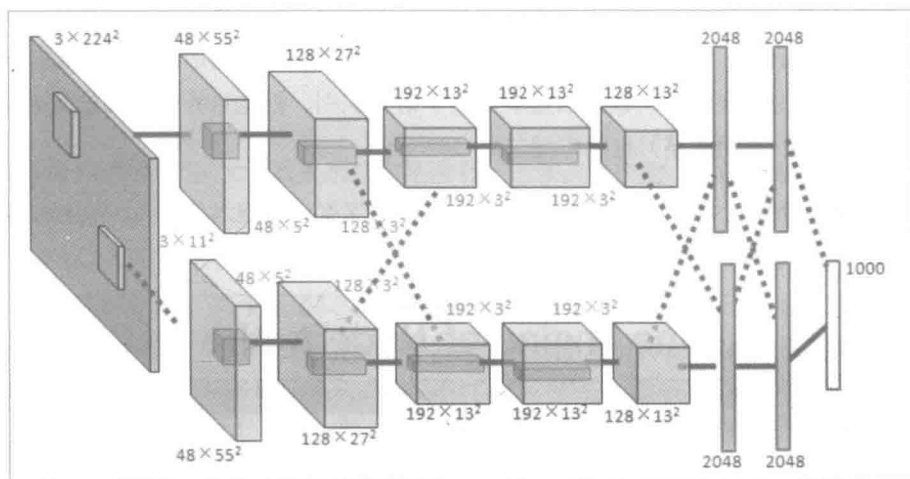


图1-2 AlexNet结构

为什么深度学习在 2012 年而不是其他时间爆发? 主要有 3 个有利因素:

- (1) 更大的数据集, 如 ImageNet。
- (2) 新的深度学习技术, 如 ReLU、Dropout 等技术。
- (3) 新的计算硬件, 如 GPU。

我们先看看国外的深度学习有哪些进展。

1.2 师夷长技

1.2.1 谷歌与微软

Google 在 Geoffrey Hinton 等大牛的带领下, 在理论与技术方面一直保持世界领先地位。利

用 GoogLeNet, 在 2014 年 ILSVRC 中其分类错误率低至 6.66%。

基础平台包括早期基于大规模 CPU 集群的 DistBelief (由 16000 个计算节点构成) 和近期开放的支持 GPU 加速的 TensorFlow。2016 年朋友圈刷屏的“阿尔法狗”(AlphaGo) 也是 Google 强大深度学习的具体案例之一。

Microsoft 在 2015 年 ILSVRC 目标检测任务中使用深度残余学习框架 (Deep Residual Learning Framework) 取得绝对优势, 赢得 200 个类目中 194 个最佳检出率, 平均检出概率高达 62% (2014 年同一任务最好结果为 37%)。基于 Caffe 实现的 Fast-RCNN (作者为 Ross Girshick) 在目标识别领域占有重要地位。

Microsoft 在基础平台方面也势头强劲, 2015 年推出的 Azure Machine Learning Studio 有大量的机器学习算法, 适合用来构建预测分析解决方案。这些算法可用于一般的机器学习, 如回归分析、分类、聚类 and 异常检测, 且每一个都可以解决不同类型的机器学习问题。为其作支撑的不仅有高可扩展性、支持 CPU/GPU 计算的 Minerva 及分布式深度学习训练系统 Adam、CNTK, 还有利用 Catapult 加速深度卷积神经网络 (DCNN) 的项目也在进行中。

1.2.2 Facebook、亚马逊与 NVIDIA

Facebook 于 2013 年成立了人工智能实验室, 在 Yann LeCun 的带领下 Facebook 同纽约大学数据科学中心在数据科学、机器学习、人工智能领域展开合作, 代表性工作有最著名的开源深度学习项目 Torch (<http://torch.ch/>) 和 fbconv (<https://github.com/facebook/fbconv>)。

Amazon 本身是做 IaaS 平台的, 看到机器学习如火如荼的发展, 也迅速融入并推出了云上的机器学习服务 (<http://aws.amazon.com/cn/machine-learning/>), 提供一种 PaaS 模式。Amazon Machine Learning 提供可视化的工具和向导, 无须学习复杂的机器学习算法和技术。使用简单的 API 即可让用户应用程序轻松获得预测能力, 而无须实现自定义预测生成码或管理任何基础设施。采用 Amazon 内部使用的机器学习方法, 非常容易扩展。而且, 使用 Amazon Machine Learning 不需要对硬件或软件事先投入资金, 只需按使用量付费。

另外, 不得不提 NVIDIA, 这家老牌显卡制造商也将未来方向瞄准了深度学习, 于 GTC 2015、2016 连续发布多款面向深度学习的 GPU 加速器硬件 (Titan X、Tesla P100)、加速库 (cuDNN) 和解决方案 (DIGITS DevBox、DGX-1), 为深度学习的普及和更大模型的支持起到推波助澜的作用。

以上为国外情况, 国内情况又如何呢?

1.3 中国崛起

1.3.1 BAT 在路上

百度是国内较早开展深度学习研究的企业，于 2013 年年初创立了百度深度学习实验室 (Institute of Deep Learning, IDL, <http://idl.baidu.com/>)，斯坦福大学教授、Google 大脑创始人 Andrew Ng 随后加入。IDL 研究方向包括深度学习&机器学习、机器人、人机交互、3D 视觉、图像识别、语音识别等，同时开展了一系列深度学习相关的创新项目，如无人机、智能自行车 DuBike、自动驾驶汽车、智能眼镜 BaiduEye 等。

百度在深度学习计算平台基础设施建设方面一直走在国内互联网公司的前列，百度在 ImageNet 挑战中取得的成绩得益于其超级计算机 Minwa (36 个服务器节点，每个节点 2 个六核 Xeon E5-2620 和 4 个 NVIDIA Tesla K40m GPU)。为了提高深度学习算法的计算速度，百度在 GPU 和 CPU 上做了很多优化，发表了一些深度学习算法 GPU 加速的论文（虽然中间有点小插曲）。经过这些工作，百度也意识到 GPU、CPU 在深度学习应用中的成本效率、能耗效率和目标间的差距。在充分考量各种芯片的特性后，可编程、低功耗并拥有超强并行计算能力的 FPGA 走进了百度工程师们的视野。百度开始尝试用 FPGA 打造 AI 专有芯片，并成就了第一版 AI 专有芯片版百度大脑——FPGA 版百度大脑。这使得百度成为了全球最早将 FPGA 规模应用在人工智能领域的公司。

阿里巴巴作为电子商务巨头，很早就看到了深度学习在商品检索方面的应用价值，在阿里巴巴图像搜索的领军人物、阿里巴巴搜索事业部研究员华先胜的带领下迅速将深度学习技术成功应用到手机淘宝图像搜索业务——拍立淘中。2015 年“双 11”当天，上千万消费者使用了拍立淘功能，引导了数千万元的销售额。拍立淘上线一年以来，所覆盖的类目范畴已经从最开始的女装，发展到目前的男女装、鞋包、配饰、食品、数码、家居、日用百货、内衣、瓶饮等十余个类目。与通用搜索主要依靠字节不同，图像搜索被主要定义为“以图搜图”。据华先胜介绍，图像搜索的第一步是训练计算机进行图像理解，也就是通过计算机将图片中的要素，包括人像、颜色、纹理等具体特征以及深度学习产生的图像描述，转换为类似于文字的“视觉词”，编成索引之后，才能再进行第二步——图像搜索。图像搜索仍有很多未知领域有待探索。在华先胜看来，能推动图像搜索下一步突破的关键有三点：深度学习、大数据分析和大量用户使用反馈。环顾国内外，似乎只有阿里巴巴能够同时具备这三个条件。对于“拍立淘”的未来，华先胜表示，拍立淘将会拓展到更多领域，力争成为人们获取信息（包括购物、教育、娱乐、新闻、知识等）的一个快捷、有趣、有效的入口，而不仅仅是搜寻商品的入口。

阿里巴巴在基础平台建设方面起步虽晚，但发展迅速，利用装备 NVIDIA Tesla GPU 的高性能计算集群，不仅完美支撑拍立淘、搜索、OCR、绿网、神马语音、iDST 等内部业务，还进一步在 2015 年 10 月 14 日云栖大会上正式宣布通过阿里云对外提供公共云上的 HPC 服务 (<https://www.aliyun.com/product/hpc>)，使普通用户也有机会享受高性能计算平台带来的高效性和便利性。目前越来越多的中小企业选择租用云端 HPC 服务器，而不是自建机房做繁杂冗长的运维工作。最新机型 G4 配备了双 Tesla M40 作为加速器，可大大提高深度学习应用的运行效率，基于 Docker 的快速环境部署大幅降低了客户使用深度学习框架的门槛，可谓开箱即用。

腾讯拥有海量的社交关系数据，在深度学习应用方面潜力巨大，目前主要应用为语音识别、图像识别和广告推荐。腾讯优图 (BestImage, <http://open.youtu.qq.com/>) 是腾讯旗下顶级的机器学习研发团队，专注于图像处理、模式识别、深度学习等方向，在人脸检测、五官定位、人脸识别、图像理解领域都积累了完整解决方案和领先的技术水平。

腾讯在深度学习基础平台方面经历多次升级逐步完善，在 Mariana 基础上针对多种应用打造出 Mariana DNN、Mariana CNN、Mariana Cluster 等基础框架，在微信语音识别、微信图像识别方面均已成功落地，在图文类效果广告点击率提升方面也取得初步的应用。

1.3.2 星光闪耀

中国科学院计算所计算机体系结构国家重点实验室未来计算组陈云霁研究员领导的团队提出了国际上首个深度神经网络处理器寒武纪 1 号 (DianNao)，通过高效的分块处理和访存优化，能高效处理任意规模、任意深度的人工神经网络，以不到通用处理器 1/10 的面积和功耗达到了 100 倍以上的神经网络处理速度，性能功耗比提升了 1000 倍。该项工作意味着，处理器结构设计的创新，有望在未来使得手机移动终端具备谷歌大脑级别的认知处理能力。2014 年 12 月，新推出的寒武纪 2 号神经网络处理器 (DaDianNao) 荣获年度 Micro 最佳论文。“DaDianNao”又有多项突破，性能继续大幅度提升，与通用芯片和 GPU 相比，计算速度提高几十倍，功耗只有十分之一，整体能效提高 450 倍。陈云霁透露，这种芯片将用在国产手机上。“寒武纪”芯片执行的是一种与通用计算完全不同的指令集——电脑语“DianNaoYu”。所谓指令集就是电脑“语言”，其直接面对大规模神经元和突触的处理，一条指令即可完成一组神经元的处理。模拟实验表明，“寒武纪”相对于传统的执行 x86 指令集的芯片，有两个数量级的性能提升。与传统的通用计算指令集相比，“电脑语”显然更类似于人类大脑的学习方式，因此有人将其称为“下一代人工智能技术”。“电脑语”被计算机体系结构领域顶级国际会议 ISCA 2016 接收，其评分在近 300 篇投稿中排名第一。陈天石研究员表示，“寒武纪”不是代替中央处理器的颠覆式革命。从目前的情况来看，它更像是一款针对智能认知等应用的专用芯片——“我们的优势主要集中在

人脸识别、声音识别等人工智能方面。比如，传统的手机或个人电脑主板上嵌入‘寒武纪’芯片后，将极大地提高处理这类任务的速度，并且降低能耗”。

科大讯飞股份有限公司 (<http://www.iflytek.com/>) 是一家专业从事智能语音及语言技术、人工智能技术研究、软件及芯片产品开发、语音信息服务及电子政务系统集成的国家级骨干软件企业。该公司的智能语音核心技术代表了世界最高水平，是我国产业化实体中在语音技术领域基础研究时间最长、资产规模最大、历届评测成绩最好、专业人才最多及市场占有率最高的公司。语音技术实现了人机语音交互，使人与机器之间的沟通变得像人与人沟通一样简单。此外，语音技术还包括口语评测、语音编码、音色转换、语音消噪和增强等技术，有着广阔的应用空间。科大讯飞作为中国最大的智能语音技术提供商，在智能语音技术领域有着长期的研究积累，并在语音合成、语音识别、口语评测、自然语言处理等多项技术上拥有国际领先的成果。

1.3.3 企业热是风向标

近两年国内利用深度学习技术的创业公司如雨后春笋般涌现。

商汤科技 SenseTime (<http://www.sensetime.com/>) 致力于引领人工智能核心“深度学习”技术突破，构建人工智能、大数据分析行业解决方案。在人工智能产业兴起的大背景下，商汤科技拥有在技术、人才、专利上多年的积累，聚集了一批出色的华人深度学习、计算机视觉科学家，以及来自于谷歌、百度、微软、联想等产业界领军人物。在人脸识别、文字识别、人体识别、车辆识别、物体识别、图像处理等前瞻性应用技术上，商汤科技均拥有核心原创技术和持续进行学术研发的潜力；在业务上，商汤集团深耕金融、移动互联网、安防监控三大行业，已与中国移动、银联、京东、拉卡拉、华为、小米、新浪微博、科大讯飞、东方网力、英伟达等知名公司开展深度合作，推动行业产品智能化升级，开拓中国原创人工智能技术更多可能。

Face++™ (<http://www.faceplusplus.com.cn/>) 是北京旷视科技 (Megvii) 有限公司旗下的新型视觉服务平台，旨在提供简单易用、功能强大、平台兼容的新一代视觉服务。Face++团队专注于研发世界最好的人脸检测、识别、分析和重建技术，通过融合机器视觉、机器学习、大数据挖掘及 3D 图形学技术，致力于将最新、性能最好、使用最方便的人脸技术提供给广大开发者和用户。通过提供云端 API、离线 SDK，以及面向用户的自主研发产品形式，将人脸识别技术广泛应用到互联网及移动应用场景中。

涂鸦 (<http://www.airtake.me/>) 成立于 2014 年，专注于云服务，致力于通过智能云为厂商提供由普通硬件转变为智能硬件的完整技术解决方案分布式架构的全球部署，每日超过 10TB 和千万人次的数据吞吐量原生图像识别与机器学习能力。涂鸦在云计算技术、硬件生产以及海

外市场运营方面有丰富的经验。涂鸦在技术上获得了阿里云和亚马逊云服务的支持。

格灵深瞳 (<http://www.deepglint.com/>) 成立于 2013 年年初, 是全球第一家采用三维计算机视觉技术, 将人工智能应用于商业领域的科技公司, 致力于让计算机像人一样看懂这个世界, 并且把这一技术率先用在了安防监控和智能交通领域。让计算机看懂世界, 是格灵深瞳的使命。因为感知技术是所有存在于真实世界里的人工智能的信息入口。近年来, 深度学习和深度视觉成为了计算机视觉领域最伟大的创新和进展。格灵深瞳是全世界最早把这些技术商业化的公司之一。结合深度学习和深度视觉技术建造视觉传感器网络, 有效地赋予视频监控、智能交通以及智能驾驶等领域全新的价值。

Dress+ (衣+, <http://www.dress-plus.com/>) 成立于 2014 年, 提供在线视觉识别技术与社交网络搜索时尚商品的服务。衣+边看边买搜索引擎是领先的商品图像特征建模方案, 基于深度学习和传统方法融合商品图像特征建模算法, 既刻画了高层语义特征, 又兼顾了底层图像特征, 大大加强了衣+引擎对同款和相似款商品的检索能力, 帮助用户快速找到感兴趣的商品图像。使用衣+独有的高效特征量化压缩算法, 在保证检索效果和原始特征基本一致的条件下可以将单条记录特征压缩到 1KB 以内, 极大地提高了搜索引擎的可扩展性。衣+高实时性搜索引擎在单机单线程条件下完成 2 千万条目的检索时间小于 1s, 通过并行优化的系统支持单机亿级条目的检索时间小于 1s。

Linkface (<http://www.linkface.cn/>) 是一家人脸识别技术研发公司, 曾取得 FDDB 人脸检测公开测试世界第一、300-W Benchmark 准确率世界第一、LFW 人脸识别准确率达 99.5% 以上等一系列成绩。Linkface 开发了基于深度学习的人脸检测创新算法, 无论孤身一人还是置身人群, 抑或是处在侧脸、遮挡、模糊等情景中, 均能进行精准检测; Linkface 可准确识别出眼睛、鼻子等人脸关键位置, 在表情不同、姿态多样、遮挡模糊等状态下均可进行精准定位; 在监控、门禁、自拍、人证比对等场景中, Linkface 的识别算法能够提供精准、便捷的识别方案。

今天的主要内容是展现国内外深度学习领域一片欣欣向荣的形势, 读到这里是否已经下定决心跟着本书一探深度学习之究竟? ! 我们明天继续~

1.4 练习题

1. 搜索深度学习领域的先驱者 Geoffrey Hinton、Yann LeCun、Youshua Bengio 三大牛的个人主页, 查看其最新动态。
2. 在 Coursera 上搜索 Andrew Ng 的机器学习课程。
3. 了解各大招聘网站上深度学习相关岗位的需求情况。

第 2 天

深度学习的过往

今天让我们缅怀历史，展望未来。

深度学习有着悠久而丰富的历史。在过去很长一段时期内备受冷落，而在最近 5 年，大规模训练数据（如 ImageNet）和高性能计算硬件（GPU）的出现，为这个领域重新提供了燃料和助推器。

2.1 传统机器学习的局限性

传统机器学习技术在处理原始形态的自然数据方面有很大的局限性。

几十年来，构建模式识别或机器学习系统需要技艺高超的工程师和经验丰富的领域专家来设计特征提取器（Feature Extractor），将原始数据（如图像的像素值）转化为合适的中间表示形式或特征向量（Feature Vector），学习子系统（通常为分类器）可以对输入模式进行检测或分类，如图 2-1 所示。

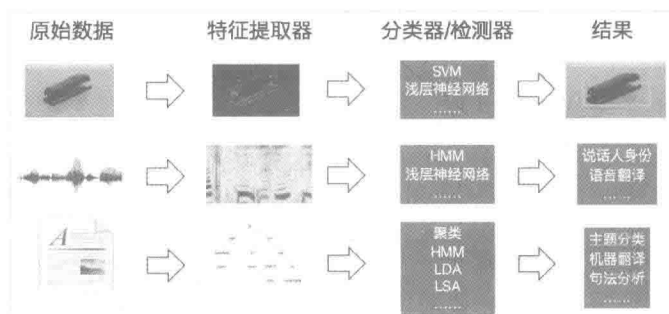


图2-1 传统机器学习处理模式

深度学习方法则不需要人工设计特征提取器，而是由机器自动学习获得，特别适用于变化多端的自然数据，具有非常优良的泛化能力和鲁棒性。

深度学习不是一天建成的，有一个长期进化的过程，我们来一探究竟。

2.2 从表示学习到深度学习

在表示学习（Representation Learning）系统中，直接以原始数据形式提供机器输入，自动发现用于检测和分类的表示（Representation）。深度学习是一种多层表示学习方法，用简单的非线性模块构建而成，这些模块将上一层表示（从原始数据开始）转化为更高层、更抽象的表示。当一个学习系统由足够多这样简单的非线性模块构建时，可以学习非常复杂的功能。

对于分类问题，高层表示能强调重要的类别信息，同时抑制无关的背景信息。一幅图像总是以像素值数组形式提供网络输入，第一层学习到的特征为边缘信息，即图像某个位置是否存在特定朝向的边缘；第二层检测边缘信息按特定方式排列组成的基本图案，而不关心边缘位置的变化；第三层将基本图案组合起来，对应典型物体的部件，后续层检测由部件组成的物体，如图 2-2 所示。深度学习最关键的方面是这些特征层不是由专家设计的，而是使用通用学习方法自动从数据学习得到。这些从低到高的“表示”是人类无法预估的，完全由机器决定哪些特征是自己需要的，哪些是可以抑制的。

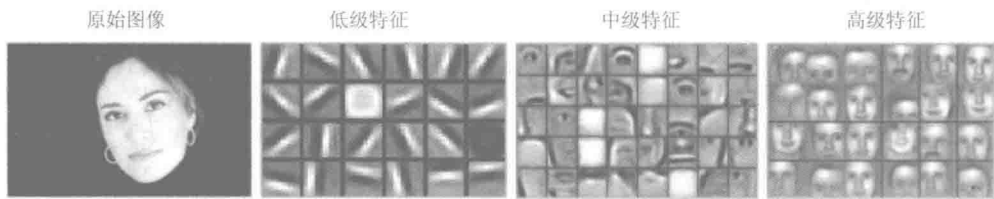


图2-2 表示学习中间特征

深度学习十分擅长在高维数据中发现复杂结构，可应用于科学、商业和政务等很多领域，在图像识别、语音识别中打破多项纪录。更惊人的是，在自然语言理解（特别是主题分类、句法分析、问答系统和语言翻译）中产生了大量有价值的结果。

深度学习只需少量人工介入，非常适合当前大规模计算系统和海量数据，在不久的将来会有更多的成功案例。当前正在开发的用于深度神经网络的新学习算法和架构将加快这个进程。

2.3 监督学习

机器学习（不论深浅）最普遍的形式是监督学习。

设想我们希望构建一个可以分类图像包含内容（房子、车、人和宠物）的系统。首先收集大量关于房子、车、人和宠物的图像数据集，每张图像都按照其类别打标签。

在训练阶段，向机器送入一张图像，产生一个得分向量，每个元素对应一个类别。我们希望真实类别在所有类别中得最高分（实际上在训练前是不会发生的）。我们通过计算目标函数来度量输出得分与期望形式得分的差异（或距离），之后机器改变内部可调参数来降低这个目标函数。可调参数为实数，常被称为权值，可被看作定义机器输入-输出函数的“旋钮”。在典型深度学习系统中，可能有上亿个可调权值，同时会有上亿个带标签的训练样本用于训练该机器。

为了恰当调节权值向量，学习算法计算梯度向量，表示每个权值增加一个微小值时目标函数的变化量，之后权值向量根据梯度向量的相反方向来调节。目标函数在所有训练样本上取平均，可看作权值向量所在高维空间的丘陵平面。负梯度向量指示当前平面最陡的下降方向，将目标函数带入距离最小值更近的地方，输出误差在平均意义上更低。

在实际中，大多数工程师都使用随机梯度下降（Stochastic Gradient Descent, SGD）方法，包括输入少量样本、计算输出和误差、计算这些样本的平均梯度、根据梯度调节权值。对训练集中大量的小样本集重复该过程，直到目标函数平均值停止下降。它之所以被称作随机梯度下降法，是由于每个小样本子集提供了所有样本平均梯度的带噪声估计。相比更精确的优化技术，这个简单方法通常能更快地找到一组好的权值。训练完成后，在一组不同的样本（称为测试集）上测试系统性能，目的是测试机器泛化能力——它在训练阶段没有见过的新输入上产生合理输出的能力。

在很多机器学习实际应用中，在人工特征上使用**线性分类器**。二分类线性分类器计算特征向量元素的加权和。如果加权和高于某个门限，输入将被分到一个特定类别。

1960 年以后，线性分类器的局限性开始被认识到，它只能将输入空间切分为非常简单的区域，即由一个超平面分离的半空间。对于像图像和语音识别这类问题，需要输入-输出函数对输入的非相关变化（位置变化、方向变化、光照变化、语音的高音和低音变化）不敏感，而对类别敏感（如白狼和萨摩耶犬）。在像素级别，两张不同姿态、不同环境下萨摩耶犬的照片会有极大的不同，而同样背景、同样位置的萨摩耶犬和白狼照片可能非常相似。对直接操作图像像素的线性分类器或其他“浅层”分类器可能不容易区分后两张照片，同时将前两张照片放在同一

类。这就是为什么浅层分类器需要好的特征提取器——有选择性地产生图片中重要类别信息的表示，同时对无关信息如姿态具有不变性——以解决选择无关困境。

为了让分类器更强大，可以使用广义非线性特征以及核函数方法。但广义特征（如高斯核函数）泛化能力差，常规方法是手动设计好的特征提取器，而这需要大量工程经验和领域专家才能完成。

如果好的特征可以使用通用学习方法自动学到，上述问题都可以避免。这是深度学习的核心优势。

一个深度学习架构是将简单模块多层堆叠，大多数模块是具备学习能力的，能计算非线性输入-输出映射。每个模块将它的输入变换，提高可选择性和表示不变性。多个非线性层（5~20层）构成的系统可以实现非常复杂的函数，例如同时做到对类间差异敏感（区分萨摩耶犬和白狼）和对类内差异不敏感（萨摩耶犬在不同背景、姿势、光照和周边物体下的照片都能正确识别）。

2.4 反向传播算法

从最早的模式识别（Pattern Recognition）时期开始，研究者的目标就是用可训练的多层网络取代人工特征工程。但该解决方案并没有被广泛认可，直到20世纪80年代中期，研究者才证明多层架构可以通过SGD训练。只要模块是其输入和内部权值的相对平滑函数，就可以使用反向传播步骤计算梯度。在20世纪70、80年代，几个不同的研究组分别独立发现该思路可行且的确可用。

利用反向传播方法计算目标函数相对多层网络权值的梯度过程，其实就是《高等数学》中求导数链式法则的工程应用。

如图2-3（左）所示为一个具有双隐层深度前馈网络的前向传播计算流程，每层我们选择其中一个节点进行计算演示。

从输入单元到第一个隐层H1计算如下：

对H1层的每个单元 j ，其值 $y_j = f(z_j)$ ， $z_j = \sum_i w_{ij} x_i$ ，其中 i 取值遍历所有输入层节点， z_j 是对前一层所有节点的加权和，这里省略了偏置项。网络中使用非线性函数 f 对 z_j 进行非线性变换，得到该层输出 y_j 。

从 H1 到 H2 计算如下：

对 H2 层的每个单元 k ，其值 $y_k = f(z_k)$, $z_k = \sum_j w_{jk} y_j$ ，其中 j 取值遍历所有 H1 层节点。

从 H2 到输出层计算如下：

对输出层的每个单元 l ，其值 $y_l = f(z_l)$, $z_l = \sum_k w_{kl} y_k$ ，其中 k 取值遍历所有 H2 层节点。

如图 2-3（右）所示为同一个深度前馈网络的反向传播计算流程，每层我们仍然选择其中一个节点进行计算演示。

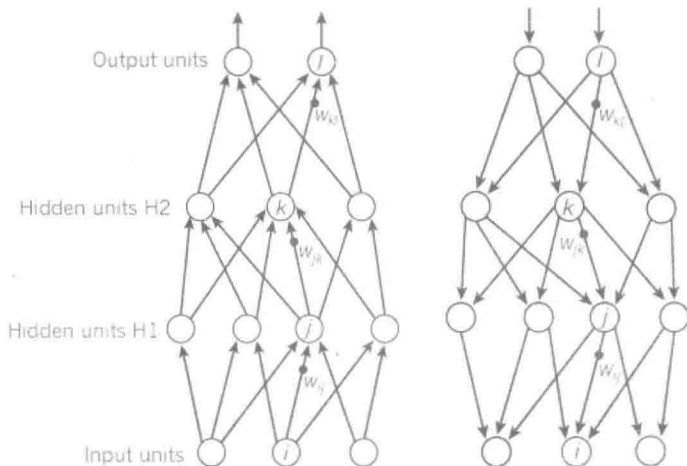


图2-3 前向传播（左）和反向传播（右）计算流程

每层首先计算相对于该层输出节点的误差梯度，即所有来自相对于后一层输入节点的误差梯度的加权和。之后使用链式法则将误差梯度传递至该层输入节点。输出单元的误差梯度通过对代价函数（或损失函数）求导得到，假设输出层单元 l 对应的代价函数项为 $E=0.5(y_l-t_l)^2$ ，其中 t_l 为期望输出值，可计算相对于 y_l 的偏导数为 y_l-t_l 。由于 $y_l=f(z_l)$ ，所以代价函数相对于 z_l 的偏导数为：

$$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l} \cdot \frac{\partial y_l}{\partial z_l} = (y_l - t_l) f'(z_l)$$

从输出单元到第二个隐层 H2 计算如下：

对 H2 层的每个单元 k ，其误差梯度为 $\frac{\partial E}{\partial y_k} = \sum_l \frac{\partial E}{\partial z_l} \cdot \frac{\partial z_l}{\partial y_k} = \sum_l w_{kl} \frac{\partial E}{\partial z_l}$ ，其中 l 取值遍历所

有输出层节点。

同理，可得出 H1 层的误差梯度为 $\frac{\partial E}{\partial y_j} = \sum_k \frac{\partial E}{\partial z_k} \cdot \frac{\partial z_k}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k} \cdot \frac{\partial y_k}{\partial z_k} \cdot w_{jk}$ ，其中 k 取遍 H2 层所有节点。

输入层的误差梯度为 $\frac{\partial E}{\partial x_i} = \sum_j \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial x_i} = \sum_j \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot w_{ij}$ ，其中 j 取遍 H1 层所有节点。

通过上面的公式可以了解到，反向传播算法的关键一点就是代价函数相对于一个模块输入的导数（或梯度），可以通过目标函数相对于该模块输出的导数反向传播求得。反向传播公式可以重复应用，将梯度从顶层输出（网络产生预测的位置）通过所有模块传递到底（输入层）。所有这些中间梯度被计算出来后，再计算代价目标函数相对于每个模块内部权值的梯度就非常容易了。

以输入层到 H1 层权值为例，其误差梯度为 $\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot x_i$ 。

2.5 卷积神经网络

有一种特殊类型的深度前馈网络，训练更简单，泛化能力比相邻层用全连接更好，这就是卷积神经网络（ConvNet）。当神经网络被抛弃时，它却在多个领域取得成功，如今在计算机视觉社区被广泛接受。

ConvNet 的四项基本原则：局部互联、共享权值、下采样以及使用多个卷积层。

共享权值意味着更少的参数量，下采样保证了局部不变性，多特征图允许不同卷积核作为不同特征提取器，训练时使用反向传播算法。典型的 ConvNet 架构如图 2-4 所示。

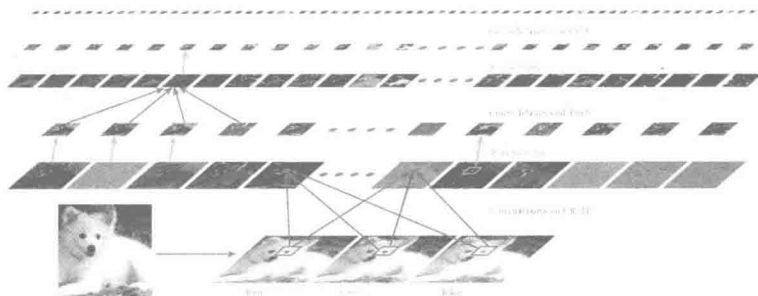


图2-4 卷积神经网络架构

前几个阶段由两种层构成：卷积层和下采样层（pooling layer）。卷积层的单元组织为特征图（feature map），每个单元通过一组称为滤波器组的权值连接到上一层特征图的局部小块。局部小块的加权和将被非线性单元（如 ReLU）处理。同一特征图的所有单元共享同一套滤波器组，而同一层的不同特征图使用不同的滤波器组。使用这种架构的原因有两方面：一方面，在一组类似图像的数据中，局部像素块具有高度相关性，形成不同的、易检测的基本图案；另一方面，图像和其他信号的局部统计具有位置无关性。

换句话说，如果一张图像存在某个基本图案，该图案可能出现在任意位置，那么不同位置单元共享相同权值可实现在数据的不同位置检测相同“模式”。一张特征图执行的滤波操作在数学上表述为离散卷积，“卷积层”名称由此而来。

卷积层的任务是检测前一层的局部连接特征，而下采样层是将语义相似的特征融合为一个。由于相对位置特征形成一个基本图案可能会有些许变化，可靠检测该图案可以使用粗粒度位置实现。

典型的下采样器计算每一张特征图（或某几张特征图）的局部小块最大值。相邻下采样器的输入源相互错开不少于一行或一列，因此可以降低表示维度，而且对平移、形变不敏感。

卷积层-非线性层-下采样层堆叠为一个基本处理栈，在一个完整的网络中可以重复多个基本处理栈后再接入更多的卷积层或全连接层。卷积网络的梯度反向传播过程与普通深度网络一样简单，所有滤波器组的权值都能得到训练。

深度神经网络揭示了很多自然信号具有复合结构的特点。高层特征可通过低层特征组合得到。图像中，棱边经过局部组合可构成基本图案，而基本图案组合成部件，部件又构成了物体。语音和文本也存在相似结构，即由声音（sound）到语音（phone），再到音素（phoneme）、音节（syllable）、单词（word）、句子（sentence）。下采样层可保证新的特征层表示不敏感于前一层元素在位置和表现上的变化。

早在 20 世纪 90 年代左右就已经有大量的卷积神经网络应用案例，从用于语音识别和文档阅读的时间延迟神经网络开始。文档阅读系统将 ConvNet 与实现语言约束的概率模型一起训练。到 90 年代末，该系统阅读全美国 10% 的支票。不久微软开发了一些基于 ConvNet 的光学字符识别（Optical Character Recognition, OCR）和手写体识别系统^[2]。ConvNet 在自然图像的目标检测（人脸和手的检测、人脸识别）中有大量实践^{[3][4][5]}。

21 世纪早期，ConvNet 已经成功应用到检测、分割和识别图像中的物体和区域。所有这些任务有相对充裕的带标签数据，例如交通标志识别、生物图像分割（特别是连接组），以及自然

图像中人脸检测、文本检测、行人检测和人体检测。近期 ConvNet 成功用于人脸检测^[6]。

更重要的是，图像可以在像素级别打标签，在技术上有所应用，包括自动驾驶机器人、自动驾驶汽车。比如 Mobileye 和 NVIDIA 准备将基于 ConvNet 的方法应用于即将发布的汽车视觉系统中。其他应用如自然语言理解和语音识别中，ConvNet 的重要性也在不断加强。

除了上述成功案例，ConvNet 被主流计算机视觉和机器学习社区所忽视，直到 2012 年的 ImageNet 比赛才改变了这一状况。那一年深度卷积网络应用到百万量级的数据集，包括 1000 个不同类别，获得优异成绩，几乎将之前最好的比赛结果错误率降低一半^[7]。GPU 的使用、ReLU 方法及一种新的规整化技术——Dropout，以及数据增强技术造就了该方案的成功，为计算机视觉领域带来巨大变革。目前几乎所有的识别和检测问题都将 ConvNet 作为主流处理方法，某些任务达到甚至超过了人类的能力^{[8][9][10]}。基于 ConvNet 的视觉系统的突出性能引无数互联网和 IT 公司竞折腰，包括 Google、Facebook、Microsoft、IBM、Yahoo!、Twitter 和 Adobe，以及国内的 BAT、迅速增长的创业公司纷纷对该技术发起研究，开发工程并部署基于 ConvNet 的图像理解产品和服务。

2.6 深度学习反思

读史使人明智，通过历史可以找到前人的闪光点，指导后人少踩坑。

深度学习也有坑。深度学习看似万能，实则有很多调参技巧在里面，掌握得当可以快速获得模型，否则可能费力不讨好。

- 模型参数远大于数据量时，相当于求解一个欠定方程，存在多解的可能性大，容易产生过拟合问题。
- 模型参数远小于数据量时，相当于求解超定方程，可能无解，或者有解但准确率很低，这属于欠拟合问题。
- 模型参数与数据量匹配时，相当于求解恰定方程，既能避免过拟合，又能兼顾准确率，但模型参数量和数据量怎样才能做到匹配，是一个工程问题。

所以，如果你选择用某个模型处理数据，那么应该考虑这个因素，越大的模型越难训练，因为需要与之匹配的数据量、一系列避免过拟合的方法才能训练得到一个较为理想的模型。幸运的是，我们可以将大模型首先在较大的数据集（如 ImageNet）上预训练，得到模型，再对特定数据集（如人脸数据）进行精调（fine-tuning），即可得到较为理想的结果。

2.7 练习题

1. 复习高等数学求导链式法则。
2. 复习线性代数矩阵乘法。

2.8 参考资料

- [1] 本部分内容参考并翻译自：Deep learning, Yann LeCun, YoshuaBengio, Geoffrey Hinton
- [2] Simard, D., Steinkraus, P. Y. & Platt, J. C. Best practices for convolutional neural networks. In Proc. Document Analysis and Recognition 958–963 (2003)
- [3] Vaillant, R., Monrocq, C. & LeCun, Y. Original approach for the localisation of objects in images. In Proc. Vision, Image, and Signal Processing 141, 245–250 (1994)
- [4] Nowlan, S. & Platt, J. in Neural Information Processing Systems 901–908 (1995)
- [5] Lawrence, S., Giles, C. L., Tsoi, A. C. & Back, A. D. Face recognition: a convolutional neural-network approach. IEEE Trans. Neural Networks 8, 98–113 (1997)
- [6] Taigman, Y., Yang, M., Ranzato, M. & Wolf, L. Deepface: closing the gap to human-level performance in face verification. In Proc. Conference on Computer Vision and Pattern Recognition 1701–1708 (2014)
- [7] <http://image-net.org/challenges/LSVRC/2012/results.html>
- [8] Surpassing Human-Level Face Verification Performance on LFW with GaussianFace, arXiv:1404.3840v1
- [9] Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, arXiv:1502.01852v1
- [10] Mastering the game of Go with deep neural networks and tree search, NATURE | VOL 529 | 28 JANUARY 2016

第 3 天

深度学习工具汇总

深度学习是一个发展迅速的领域，除了众多学术界专家在理论方面的贡献之外，工业界的贡献也不可忽视。后者将深度学习理论迅速转化为代码，应用到实际系统中，进一步结合业务优化改进，形成一股合力，推动深度学习不断进步。

今天介绍当前流行的几大深度学习工具，读者可以从中了解到每个工具的设计思路、优点和适用场景。

3.1 Caffe

Caffe (Convolutional Architecture for Fast Feature Embedding) 是由伯克利视觉和学习中心 (Berkeley Vision and Learning Center, BVLC) 开发的基于 C++/CUDA/Python 实现的卷积神经网络框架，提供了面向命令行、Matlab 和 Python 的绑定接口，项目主页见参考资料[1]。Caffe 前身为 DeCAF^[2]，作者均为贾扬清。

Caffe 标志如图 3-1 所示。



图3-1 Caffe标志

从 Caffe 的全称可以获得如下信息：

- 它实现了前馈卷积神经网络架构 (CNN)，而不是递归网络架构 (RNN)。

- 它速度快，因为利用了 MKL、OpenBLAS、cuBLAS 等计算库，支持 GPU 加速。
- 它适合做特征提取，实际上适合做二维图像数据的特征提取。

除此之外，Caffe 还具有如下特性。

- Caffe 完全开源，遵循 BSD-2 协议。
- Caffe 提供了一整套工具集，可用于模型训练、预测、微调、发布、数据预处理，以及良好的自动测试。
- Caffe 带有一系列参考模型和快速上手例程。
- Caffe 在国内外有比较活跃的社区，有很多衍生项目，如 Caffe for Windows、Caffe with OpenCL、NVIDIA DIGITS2、R-CNN 等。
- Caffe 代码组织良好，可读性强，通过掌握 Caffe 代码可以很容易学习其他框架。

以上因素使 Caffe 成为深度学习初学者入坑的首选。本书将以 Caffe 作为入口带领读者步入深度学习的大门。

3.2 Torch & OverFeat

Torch^[3]是一个出现较早的支持大部分机器学习算法的科学计算框架，从 2000 年第一个版本开始，目前已经发布了 4 个版本（Torch 1、Torch 3、Torch 5、Torch 7）。Torch 使用轻量脚本语言 Lua 及其 C/CUDA 扩展模块实现，底层数值计算通过高效的 OpenMP/SSE/CUDA 加速，同时具备灵活性和速度优势。得益于 Lua 的轻量接口，Torch 可以很容易接入第三方软件。

Torch 标志如图 3-2 所示。



图3-2 Torch标志

Torch 为机器学习提供了类似于 Matlab 的环境，目前纽约大学（NYU）、Facebook AI 实验室和 Google DeepMind Torch 均使用该框架做深度学习研究。Torch 不仅支持 CPU/GPU 上运行，

甚至支持嵌入式设备如 iOS、Android、FPGA。

Torch 完全开源，遵循 BSD 协议。目前 Torch 7 带有 8 个内置包：

- torch——Torch 7 的主包，提供 Tensors 基本数据类型和操作、简单的序列化接口和其他基本功能。
- lab & plot——提供标准的类似于 Matlab 的函数，用于创建、变换、打印 Tensors。
- qt——Qt 和 Lua 的完全绑定，实现 Torch 7 Tensors 和 QImage 之间的透明转换，美观的图形界面非常适合快速开发交互式演示程序。
- nn——提供一组标准神经网络模块，以及一组容器模块，可用于定义任意有向（无环或有环）图。显式描述图结构，使用可插入模块，避免了复杂的图解析器，或者其他中间件编译器。下面的例子用 Torch.nn 实现了 MLP：

```
// mlp_test.lua
require 'nn'
mlp = nn.Sequential()
mlp:add(nn.Linear(100, 1000))
mlp:add(nn.Tanh())
mlp:add(nn.Linear(1000, 10))
mlp:add(nn.SoftMax())
print(mlp)
```

- image——图像处理包，提供了所有标准图像处理函数（如载入/保存图像、缩放/旋转、色彩空间变换、卷积、高斯核等）。
- optim——提供最陡下降法、共轭梯度法和有限内存下 BFGS 等优化算法包。
- unsup——包括几个非监督学习算法，如 K-means、稀疏编码、自动编码器。
- third-party——在上述包的基础上进一步封装不断增加的便捷软件包。

OverFeat^[4]是一个在 ImageNet 数据集中使用 Torch 7 训练的特征提取器，实现了图像识别、定位和检测三位一体的集成系统，取得了 ILSVRC 2013 定位任务冠军，同时在分类和检测任务中也取得了不错的成绩。

3.3 MxNet

MxNet^[5]是一个面向效率和灵活性设计的深度学习框架，吸收了多种不同框架（Minerva/Torch 7/Theano）的优点，加入了更多新的功能，如更加方便的多卡和多机分布式运行，目前 MxNet 比 cxxnet 快 40%，而且 GPU 显存使用少了一半。

MxNet 标志如图 3-3 所示。



图3-3 MxNet标志

MxNet 提供了两种编程接口：

- N 维数组（ndarray）接口，类似于 Matlab 或 Python 中的 `numpy.ndarray` 或 `torch.tensor`。它独有的优势在于通过背后的 engine 可以在性能和内存使用上优于其他框架；
- 符号（symbolic）接口，可以快速构建一个神经网络，实现自动求导功能。

目前 MxNet 还在快速发展中，以后目标是更多的语言绑定（目前支持比较好的是 Python，马上会有 Julia 和 R）、更好的文档和更多的应用（语言建模、语音、机器翻译、视频）发展。

3.4 TensorFlow

Google 于 2011 年推出了人工深度学习系统——DistBelief^[6]。通过 DistBelief，Google 能够扫描数据中心数以千计的核心，并建立更大的神经网络。这个系统将 Google 应用中的语音识别率提高了 25%，以及在 Google Photos 中建立了图片搜索，并驱动了 Google 的图片字幕匹配实验。DistBelief 还存在不少不足和限制。它很难被设置，和 Google 内部的基础设施联系也过于紧密，这导致研究代码几乎不可能分享。

针对以上问题，Google 在 2015 Google Research Blog 宣布推出新一代人工智能学习系统——TensorFlow^[7]。TensorFlow 标志如图 3-4 所示。

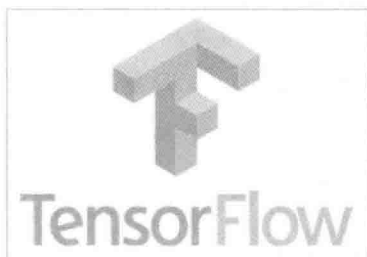


图3-4 TensorFlow标志

TensorFlow 是一个异构分布式系统上的大规模机器学习框架，移植性好（小到移动设备如手机，大到大规模集群，都能支持），支持多种深度学习模型。根据 Google 的说法，TensorFlow 是综合的、灵活的、可移植的、易用的，更为关键的是，它是开源的。与此同时，TensorFlow 的速度相比前代的 DistBelief 有了不小的提升，在一些跑分测试中，TensorFlow 的得分是第一代系统的两倍。尽管如此，但从 3.6 节的对比结果来看，TensorFlow 的效率仍然比不过其他大部分开源框架。不过，随着 TensorFlow 源码逐步开放，对新硬件、新设备、新的加速库如 cuDNN 的支持力度不断提升，其成为目前极具潜力的深度学习框架，读者可以在掌握 Caffe 后继续深入研究 TensorFlow。

TensorFlow 计算流如图 3-5 所示。

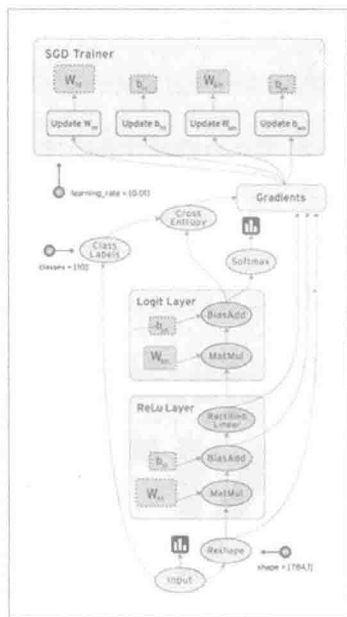


图3-5 TensorFlow计算流
www.aibbt.com 让未来触手可及

3.5 Theano

Theano^[8]是由 LISA 开发的基于 Python 的深度学习框架，可以定义数学表达式并高效地优化、求值。

Theano 标志如图 3-6 所示。



图3-6 Theano标志

Theano 支持机器学习中的逻辑回归 (Logistic Regression, LR)、多层感知器 (MultiLayer Perceptron, MLP)、深度卷积网络等监督学习方法，以及自编码器 (Auto Encoder, AE)、降噪自编码器、限制玻尔兹曼机 (Restricted Boltzman Machine, RBM)、深度置信网络 (Deep Belief Network, DBN) 等非监督/半监督学习方法，在国外教育领域非常受欢迎，一些机器学习课程都是采用 Theano 教学的。但是 Theano 有个致命短板，就是计算速度慢，虽然有 GPU 加速，但仍然不如其他框架高效，所以只适合研究人员使用，不适合在线上环境部署。

3.6 CNTK

CNTK (Computational Network Toolkit)^[9]是微软推出的开源深度学习框架，通过一系列计算步骤构成有向图来表达网络。

CNTK 标志如图 3-7 所示。



图3-7 CNTK标志

CNTK 的优点是高性能、高灵活性、可扩展性好。CNTK 支持 CNN、LSTM、RNN 等流行网络结构，支持分布式训练。在纯 CPU、单 GPU、多 GPU、多机多 GPU 硬件平台下都具有较高的性能。

从图 3-8 中看到，CNTK 支持双机 8 GPU 并行处理，而其他框架只支持单 GPU（Theano）或单机多 GPU（TensorFlow、Torch 7、Caffe）。从单 GPU 对比性能来看，Theano 是性能最低的，而 CNTK、Torch 7、Caffe 相差不大。单机 4 GPU 的性能对比结果显示了 CNTK 具有极高的效率。GitHub 上也有对常见的深度学习框架卷积计算性能的对比情况^[10]。

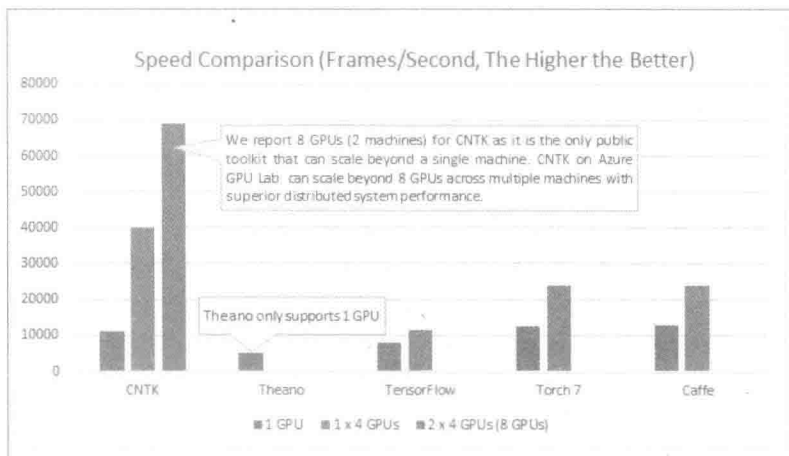


图3-8 CNTK提供的性能数据

虽然 CNTK 有上述优点，但同时要看到微软对自家 Windows 支持不遗余力，导致 CNTK 对 Windows 平台支持最好，不推荐作为深度学习初学者入门工具，而是选择拥有更大社区的 Caffe。

除了上述深度学习工具，还有一些特殊的工具作为选择，具体情况可参见附录 A。

掌握了今天的内容并深入研究、仔细评测、对比不同框架，理解内在的设计理念，深入对比各自的优势和不足，你会成长为优秀的深度学习平台工程师。

3.7 练习题

1. 到上述工具的主页查看最新动态。
2. 思考：深度学习工具为什么总是发源于国外？

3. 如果将来你想写一个深度学习框架，需要考虑哪些因素？

3.8 参考资料

- [1] BVLC Caffe 项目主页 <http://caffe.berkeleyvision.org/>
- [2] DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition, <http://arxiv.org/abs/1310.1531>
- [3] Torch 主页 <http://torch.ch/>
- [4] OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks, <http://arxiv.org/abs/1312.6229v4>
- [5] MxNet, <https://github.com/dmlc/mxnet>
- [6] Large Scale Distributed Deep Networks, NIPS 2012
- [7] TensorFlow, <http://tensorflow.org/>
- [8] Theano, <http://www.deeplearning.net/software/theano/>
- [9] CNTK 主页 <http://www.cntk.ai/>
- [10] <https://github.com/soumith/convnet-benchmarks>

第4天

准备 Caffe 环境

今天的主要任务是准备 Caffe 运行环境^[1]，确切地说，是准备本书上篇、中篇的 Caffe 环境，假设读者手头（至少）有一台安装了 Linux/Windows/Mac OS 的笔记本电脑。考虑到大部分初学者条件有限，为了快速入门，可以先学习在 CPU 上运行 Caffe 并阅读其中的 C++代码，而不是将大量精力浪费在重装操作系统、更新 GPU 驱动、阅读晦涩的 CUDA 代码上。对于希望使用 GPU 加速计算的读者，可以参考第 15 天内容。

4.1 Mac OS 环境准备

笔者写作时使用 Mac OS 作为代码编辑和阅读环境，笔记本电脑配置如图 4-1 所示。



图4-1 CPU模式Caffe运行环境

Mac OS 的命令行和 Linux 发行版几乎一样。CPU 模式的 Caffe 在 Mac 上的编译过程如下。

- (1) 安装 homebrew 包管理工具，作用相当于 yum 或 apt-get。

```
$ ruby -e "$(curl -fsSL\
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

(2) 等待安装成功，然后利用该工具安装 Caffe 依赖包：

```
$ brew install -vd snappy leveldb gflags glogs zip lmdb
$ brew tap homebrew/science
$ brew install hdf5 opencv
$ brew install protobuf boost wget
```

(3) 下载 Caffe 源码：

```
$ git clone https://github.com/bvlc/caffe.git
$ cd caffe/
$ mv Makefile.config.example Makefile.config
```

(4) 修改 Makefile.config，打开 CPU_ONLY 选项，保存。

```
# 仅 CPU 模式开关，打开该选项（去掉“#”）表示 Caffe 编译时仅支持 CPU，不支持 GPU
CPU_ONLY := 1
```

(5) 执行 make 进行编译：

```
# -j 选项表示使用多线程编译，利用所有可用的 CPU，加快编译速度
# 也可指定数字，如-j8 表示开启 8 个线程编译
$ make -j
```

等待编译成功。如果编译报错，请结合 4.5 节进行排查。

4.2 Ubuntu 环境准备

在 Ubuntu 14.04 系统中，Caffe 的所有依赖包都可以使用 apt-get 大法搞定。

```
# 在 Ubuntu 下如果没有使用 root 账号，则每个命令前需要加 sudo
$ sudo apt-get install git
$ sudo apt-get install libprotobuf-dev libleveldb-dev libsnappy-dev libopencv-dev
libhdf5-serial-dev protobuf-compiler
$ sudo apt-get install --no-install-recommends libboost-all-dev
$ sudo apt-get install libatlas-base-dev
$ sudo apt-get install python-dev
$ sudo apt-get install libgflags-dev libgoogle-glog-dev liblmdb-dev
```

(1) 下载 Caffe 源码:

```
$ git clone https://github.com/bvlc/caffe.git
$ cd caffe/
$ mv Makefile.config.example Makefile.config
```

(2) 同 4.1 节, 修改 Makefile.config, 打开 CPU_ONLY 选项, 保存。

```
$ make -j
```

等待编译成功。如果编译报错, 请结合 4.5 节进行排查。

4.3 RHEL/Fedora/CentOS 环境准备

通用依赖可以这样安装:

```
$ sudo yum install protobuf-devel leveldb-devel snappy-devel opencv-devel boost-devel
hdf5-devel atlas-devel
```

剩下的依赖, 对于最新系统可以这样安装:

```
$ sudo yum install gflags-devel glog-devel lmdb-devel
```

(1) 下载 Caffe 源码:

```
$ git clone https://github.com/bvlc/caffe.git
$ cd caffe/
$ mv Makefile.config.example Makefile.config
```

(2) 同 4.1 节, 修改 Makefile.config, 打开 CPU_ONLY 选项, 保存。

```
$ make -j
```

等待编译成功。如果编译报错, 请结合 4.5 节进行排查。

4.4 Windows 环境准备

考虑到很多读者都使用 Windows 平台, 所以增加了该节内容。Microsoft 为 Windows 用户提供了一套 Caffe for Windows 分支^[2]。

○ 操作系统: 推荐 Windows Server 2012 R2 64bit 或 Windows 7 SP1 64bit 以上

○ 编译环境 (必选): Visual Studio 2013 Ultimate 版^[3]
www.aibbt.com 让未来触手可及

由于目前只需编译 CPU 模式 Caffe，故暂时不需要安装 CUDA Toolkit 7.5^[4]和 cuDNN^[5]。

(1) 安装 Visual Studio 2013。将 Microsoft/caffe 代码下载到本地磁盘，本文路径为 C:\Users\Administrator\Desktop\caffe-master。

(2) 进入 C:\Users\Administrator\Desktop\caffe-master\windows 目录，将文件 CommonSettings.props.example 重命名为 CommonSettings.props，修改其内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003"><ImportGroup Label="PropertySheets" />
<PropertyGroup Label="UserMacros"><BuildDir>$(SolutionDir)..\Build</BuildDir>
<!--NOTE: CpuOnlyBuild and UseCuDNN flags can't be set at the same time.-->
<CpuOnlyBuild>true</CpuOnlyBuild>
<UseCuDNN>false</UseCuDNN>
<CudaVersion>7.5</CudaVersion>
..... (后面的内容都不用改，此处略去)
```

(3) 修改完成后，保存该文件。双击同一目录下的 Caffe.sln 文件，打开 Windows Caffe 工程。

(4) 单击菜单“生成”→“重新生成解决方案”，开始漫长的编译过程。

在预编译阶段，Visual Studio 2013 会通过 NuGet 工具自动获取预编译的 Caffe 依赖包，放置于 C:\Users\Administrator\Desktop\NugetPackages 下。打开该目录，查看下载好的依赖包，如图 4-2 所示。

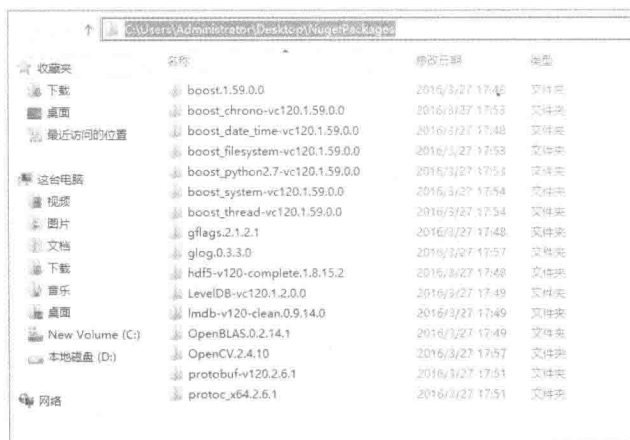


图4-2 查看依赖包

可见，与 Linux 下的依赖并无二致。查看当前目录属性，如图 4-3 所示。



图4-3 查看当前目录属性

占用空间大约 1GB，同样的依赖包在 Linux 下只有不到 200MB。

编译成功后，生成的可执行文件和库位于 C:\Users\Administrator\Desktop\caffe-master\Build\x64\Release 下，如图 4-4 所示。

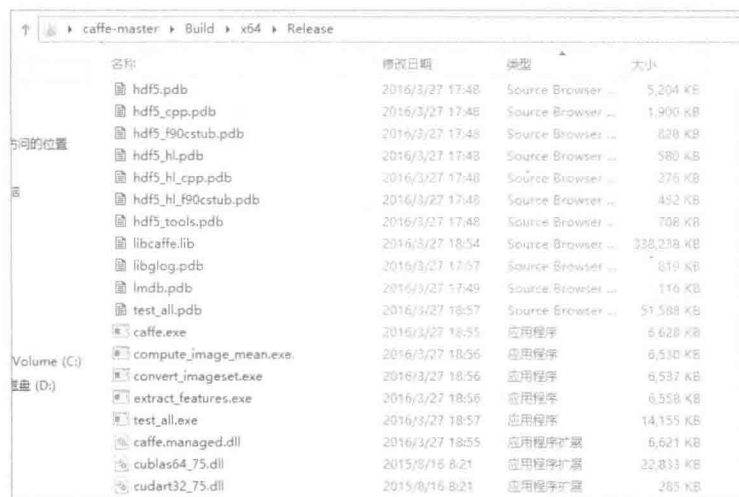


图4-4 查看生成的可执行文件和库
www.aibbt.com 让未来触手可及

看到了生成的可执行程序，其中 `caffe.exe` 我们后面会经常用到。笔者提供一份编译好的 Windows Caffe 工程^[6]，读者可下载到本地直接运行。

4.5 常见问题

一些早期的 Linux 发行版（如 Ubuntu 12.04、CentOS 6.5 等）可能找不到 `glog`、`gflags`、`lmdb` 三个依赖包，此时需要以源码编译方式安装，步骤如下：

```
# glog
$ wget https://google-glog.googlecode.com/files/glog-0.3.3.tar.gz
$ tar zxvf glog-0.3.3.tar.gz
$ cd glog-0.3.3
$ ./configure
$ make && make install

# gflags
$ wget https://github.com/schuhschuh/gflags/archive/master.zip
$ unzip master.zip
$ cd gflags-master
$ mkdir build && cd build
$ export CXXFLAGS="-fPIC" && cmake ..&& make VERBOSE=1
$ make && make install

# lmdb
$ git clone https://github.com/LMDB/lmdb
$ cd lmdb/libraries/liblmdb
$ make && make install
```

在比较旧的系统上准备 Caffe 环境稍微复杂些。推荐初学者在阅读上篇、中篇时使用 Ubuntu/Windows/Mac OS，不需要配备 GPU 和 CUDA 环境，避免第一步就陷入编译困境。随着对 Caffe 了解的深入，以及心智的成熟，读者自然会从容处理这些问题。

4.6 练习题

1. 试着用 Linux 的包管理器 `yum` 或 `apt-get` 安装 `python-dev`、`python-pip`、`numpy`。
2. 用 `yum` 或 `apt-get` 安装的软件包都在什么位置？
3. 如果有两台同样系统的机器，其一台安装了某依赖包，另一台没有安装，能否将已安装

依赖包的机器环境复制到第另一台机器上？

4.7 参考资料

[1] <http://caffe.berkeleyvision.org/installation.html>

[2] <https://github.com/microsoft/caffe>

[3] Visual Studio 2013 Ultimate 版免费获取地址: http://download.microsoft.com/download/9/3/E/93EA27FF-DB02-4822-8771-DCA0238957E9/vs2013.5_ult_chs.iso?type=ISO

[4] <https://developer.nvidia.com/cuda-downloads>

[5] <https://developer.nvidia.com/cudnn>

[6] 百度云盘 (<http://pan.baidu.com/s/1sks4XFv>, 提取码: idi7)

第5天

Caffe 依赖包解析

完成了基础环境准备工作，可能读者还有疑问，那些命令到底都安装了些什么包？到底有什么用？今天将揭开谜底。建议有 C++ 基础的读者深入阅读，了解一些开源库以节省开发时间。如果对依赖包不感兴趣，可直接跳过这部分内容，遇到问题时再回过头来阅读亦可。

为了方便读者实验，今天用到的所有源码都可以从笔者的百度云盘^[1]获得。

5.1 ProtoBuffer

ProtoBuffer 是由 Google 开发的一种可以实现内存与非易失存储介质（如硬盘文件）交换的协议接口。Caffe 源码中大量使用 ProtoBuffer 作为权值和模型参数的载体。一般开发者对参数管理各有好恶，有人喜欢 TXT 的易于修改，有人喜欢 BIN 的读写高效，也有人喜欢图形化配置的直观形象。不一致的参数管理带来很多问题，例如，一个项目组内不同成员必须约定一套统一的参数方案，或者称为通信协议，才便于模块集成。ProtoBuffer 工具完美地解决了这个问题，用户只需要建立统一的参数描述文件（proto），然后利用 `protoc` 编译就能让协议细节等关键部分代码自动生成，节省了大量的开发、调试时间。使用 ProtoBuffer 还可以跨语言（C++/Java/Python）传递相同的数据结构，让团队协作更有效率。

注意：有时旧版本的 ProtoBuffer 生成的文件在新版本中使用会有各种不易排查的错误信息，所以推荐在需要运行 Caffe 的环境下都使用同一版本 ProtoBuffer。

从笔者的百度云盘^[1]下载安装文件后解压：

```
$ tar zxvf protobuf-2.5.0.tar.gz
$ cd protobuf-2.5.0
$ ./configure --prefix=/home/yourname/local_install/
```

请注意，我们并没有将 Protobuf 安装到系统默认目录/usr/或/usr/local/下，而是安装到本地目录/home/yourname/local_install/下，这样做的好处是便于迁移。在一台机器上安装好的 Caffe 及其依赖，能迅速迁移到另一台机器上而无需重复编译、安装。后面所有第三方依赖软件包都会安装到这个目录下。

```
$ make
$ make install
```

这样就完成了 Protobuf 软件包的安装。为了检验是否安装成功，看一下安装目录：

```
$ ls ~/local_install/bin/
protoc
```

看到了 protoc 可执行文件，说明安装成功。为了能在命令行运行，我们将该目录加入 PATH 中：

```
$ export PATH=~/local_install/bin/:$PATH
```

可以将其写入/home/yourname/.bashrc，以实现建立会话时自动配置环境。接下来进入 Caffe 根目录，修改 Makefile.config，在 INCLUDE_DIRS 后面加入~/local_install/include，在 LIBRARY_DIRS 后面加入~/local_install/lib。

看到这里，可能很多读者都知道了如何用 apt 以及源码编译方式安装 ProtoBuffer，但仍然不太理解 ProtoBuffer 的具体用法，下面给出一个简单的例子来帮助理解。

在 Caffe 源码框架中找到 models/bvlc_reference_caffenet/solver.prototxt 文件，用 vi 打开，会看到如下文本内容：

```
net: "models/bvlc_reference_caffenet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "models/bvlc_reference_caffenet/caffenet_train"
solver_mode: GPU
```

这里面记录了一些模型训练所需的超参数 (Hyper-Parameter)，用 Caffe 训练时会首先读取该文件，获得其中特定字段的数值，并据此设置内存中模型训练时的超参数变量值，从文件读取到内存的过程就是由 **ProtoBuffer** 工具协助完成的。下面我们写一个简单的测试程序。

从编译好的 Caffe 目录下找到 **build** 目录，并查看生成的 **ProtoBuffer** 相关 API 文件：

```
$ ls build/src/caffe/proto/
caffe.pb.cc  caffe.pb.d  caffe.pb.h  caffe.pb.o  caffe.pb.o.warnings.txt
```

其中，**caffe.pb.h** 和 **caffe.pb.cc** 就是用于解析 Caffe 参数配置文件、将模型权值序列化/反序列化到磁盘的协议接口。我们编写测试程序如下：

```
#include "caffe.pb.h"
#include <google/protobuf/io/coded_stream.h>
#include <google/protobuf/io/zero_copy_stream_impl.h>
#include <google/protobuf/text_format.h>

#include <iostream>

using namespace std;
using google::protobuf::io::FileInputStream;
using google::protobuf::io::FileOutputStream;
using google::protobuf::io::ZeroCopyInputStream;
using google::protobuf::io::CodedInputStream;
using google::protobuf::io::ZeroCopyOutputStream;
using google::protobuf::io::CodedOutputStream;
using google::protobuf::Message;
#include <fcntl.h>

int main(void)
{
    const char * filename = "solver.prototxt";
    caffe::SolverParameter solver_param;
    int fd = open(filename, O_RDONLY);
    if (fd == -1)
    {
        cout << "File not found: " << filename << endl;
    }
}
```

```

FileInputStream* input = new FileInputStream(fd);
bool success = google::protobuf::TextFormat::Parse(input, &solver_param);
delete input;
close(fd);
cout<<"Solver Mode = "<<solver_param.solver_mode()<<endl;
cout<<"Device id = "<<solver_param.device_id()<<endl;
cout<<"Solver Type = "<<solver_param.solver_type()<<endl;
cout<<"Random Seed = "<<solver_param.random_seed()<<endl;
// cout<<"Train net param = "<<solver_param.train_net_param()<<endl;
cout<<"Max iter = "<<solver_param.max_iter()<<endl;
cout<<"Test interval = "<<solver_param.test_interval()<<endl;

cout<<"End"<<endl;
}

```

将该程序保存为 `get_param_from_proto.cpp` 文件，用 `g++` 编译该文件，命令为：

```

$ g++ -o test get_param_from_proto.cpp ./build/src/caffe/proto/caffe.pb.cc \
-I./build/src/caffe/proto/ -I~/local_install/include -L~/local_install/lib -lprotobuf

```

运行时，将 `solver.prototxt` 复制一份，放在与 `test` 同一个目录下，运行：

```

$ ./test
Solver Mode = 1
Device id = 0
Solver Type = 0
Random Seed = -1
Max iter = 450000
Test interval = 1000
End

```

从上面程序可以看出，关键代码只有三行，就能将 `solver.prototxt` 中的配置参数按照 `caffe.proto` 的协议解析并加载到内存变量 `solver_param` 中，实现简单、高效的参数同步。在简单的背后，是 `ProtoBuffer` 自动完成了复杂的接口实现。读者可以阅读 `caffe.proto` 中的 `SolverParameter` 协议、`caffe.pb.h` 和 `caffe.pb.cc`，了解具体细节。如果采用自己动手设计协议方式来加载参数，很可能会随着版本的变化，维护越来越艰难。利用该工具，使得 `Caffe` 具有灵活性好、可扩展性强的特点。

5.2 Boost

学过 C++ 的同学应该都知道 Boost 库，它是一个功能强大、构造精巧、跨平台、开源且免费的库，被称为“C++ 准标准库”，使用了很多现代编程技术，内容涵盖字符串处理、正则表达式、容器（不是 Docker）和数据结构、并发编程、函数式编程、泛型编程、设计模式实现等诸多领域，使得 C++ 开发更加灵活、高效。更多细节请参考 <http://www.boost.org/>。

在 Caffe 中主要使用了 Boost 中的智能指针，其自带引用计数功能，可避免共享指针时造成内存泄漏或多次释放。另外，pycaffe 使用 Boost Python 实现 C/C++ 和 Python 语言的连接，方便 Python 调用 C/C++ 设计的模块。

下载 boost_1_56_0.tar.bz2 并解压：

```
tar jxvf boost_1_56_0.tar.bz2
cd boost_1_56_0/
```

然后运行：

```
./bootstrap.sh --with-libraries=system,thread,python
./b2
```

不再是典型的 configure、make、make install 三部曲，生成的库需要手动复制到安装目录下：

```
cp -r boost/ /home/yourname/local_install/include/
cp stage/lib/* /home/yourname/local_install/lib/
```

5.3 GFLAGS

下载 gflags-2.1.1.zip 并安装：

```
unzip gflags-2.1.1.zip
cd gflags-2.1.1/
mkdir build; cd build/
cmake ..
ccmake ..
```

在这里，会弹出 CMAKE 配置界面，显示如下：

```
BUILD_PACKAGING      OFF
```

```

BUILD_SHARED_LIBS          ON
BUILD_TESTING               OFF
BUILD_gflags_LIB            ON
BUILD_gflags_nothreads_LIB  ON
CMAKE_BUILD_TYPE            Release
CMAKE_INSTALL_PREFIX        /home/yourname/local_install

```

```

BUILD_PACKAGING: Enable build of distribution packages using CPack.
Press [enter] to edit option                                CMake Version 2.8.11
Press [c] to configure
Press [h] for help      Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)

```

上面两个粗体字位置需要修改,其他不变。修改完成后,先按 C 键,再按 G 键,生成 Makefile。

```

make
make install

```

GFLAGS 在 Caffe 中主要起到命令行参数解析的作用,这与 ProtoBuffer 功能类似,只是参数输入源不同。GFLAGS 的使用方法可参考 Caffe 源码中的 tools/caffe.cpp。

5.4 GLOG

GLOG 库是 Google 开发的用于记录应用程序日志的实用库,提供基于 C++ 标准输入输出流形式的接口,记录时可选择不同的日志级别,方便将重要日志和普通日志分开。下载 glog-0.3.3.tar.gz 并解压,然后编译:

```

$ tar zxvf glog-0.3.3.tar.gz
$ cd glog-0.3.3/
$ ./configure --prefix=/home/yourname/local_install/
$ make
$ make install

```

GLOG 在 Caffe 中主要起到记录日志的作用,便于开发者查看 Caffe 训练中产生的中间输出,并根据这些信息决定如何调整参数来控制收敛。从日志文件我们能非常方便地看到程序运行的流程,便于跟踪源码、定位问题。GLOG 的使用方法可参考 Caffe 源码中的 tools/caffe.cpp。

5.5 BLAS

卷积神经网络中用到的数学计算主要是矩阵、向量的计算，Caffe 中调用了 BLAS (Basic Linear Algebra Subprograms, 基本线性代数子程序) 中的相应方法。最常用的 BLAS 实现有 Intel MKL、ATLAS、OpenBLAS 等，Caffe 可以选择其中任一种。打开 Makefile.config, 找到如下几行:

```
# 选择 BLAS (基本线性代数库), 本文使用 OpenBLAS, 故修改为: open
# atlas 用于 ATLAS (默认值)
# mkl 用于 MKL
# open 用于 OpenBLAS
BLAS := open
```

这里我们选择开源且高效的 OpenBLAS 实现, 另外两种作为练习。

(1) 下载 OpenBLAS 并解压:

```
tar zxvf OpenBLAS-0.2.14.tar.gz
cd OpenBLAS-0.2.14/
```

(2) 直接编译:

```
make -j
```

(3) 安装:

```
make PREFIX=/home/yourname/local_install install
```

OpenBLAS 在 Caffe 中主要负责 CPU 端的数值计算 (如矩阵乘法)。由于调用量相当大, 该库的性能直接影响 Caffe 的运行性能。如果你已经购买了 MKL, 则不必再用 OpenBLAS。另外, 在 GPU 端的数值计算则由对应的 cuBLAS 完成, 其 API 接口与 OpenBLAS 类似。

下面介绍常用的两个函数, 位于 Caffe 源码 include/caffe/util/math_functions.hpp 中。

```
template <typename Dtype>
void caffe_cpu_gemm(const CBLAS_TRANSPOSE TransA,
    const CBLAS_TRANSPOSE TransB, const int M, const int N, const int K,
    const Dtype alpha, const Dtype* A, const Dtype* B, const Dtype beta,
    Dtype* C);
```

gemm 表示基本矩阵-矩阵乘积运算, 实现操作为: $C = \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ 。

其中, CBLAS_TRANSPOSE 是一个枚举常量, 在/home/yourname/local_install/cblas.h 中有定义:

```
typedef enum CBLAS_TRANSPOSE {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113,
CblasConjNoTrans=114} CBLAS_TRANSPOSE;
```

TransA 取值可以实现四种 op: {原矩阵 A, 转置矩阵 A^T , 共轭转置矩阵 A^H , 共轭矩阵 \bar{A} }。

M、N、K 为矩阵的维度信息。众所周知, 两个相乘的矩阵必须满足一个条件: 前矩阵的列数与后矩阵的行数相等, 这里等于 K。最终相乘的结果 C 维度为 $M \times N$, 这样就很容易得到 A、B 的维度信息了。

Caffe 用上面的函数重新包装了 OpenBLAS 的 dgemm、sgemm 函数, 简化了参数设置, 减轻了算法实现负担。

```
template <typename Dtype>
void caffe_cpu_gemv(const CBLAS_TRANSPOSE TransA, const int M, const int N,
    const Dtype alpha, const Dtype* A, const Dtype* x, const Dtype beta,
    Dtype* y);
```

gemv 为基本矩阵-向量乘积运算, 实现操作为: $y = \alpha * \text{op}(A) * x + \beta * y$ 。

其中, TransA 意义同上。矩阵 $\text{op}(A)$ 的维度为 $M \times N$, 向量 x 的维度为 $N \times 1$, 向量 y 的维度为 $M \times 1$ 。

其他数学计算函数都相对简单, 读者可以自行阅读, 本书不再逐一细述, 后面用到时会说明。

5.6 HDF5

HDF (Hierarchical Data File) 是美国国家高级计算应用中心 (NCSA) 为了满足各种领域研究需求而研制的一种能高效存储和分发科学数据的新型数据格式。它可以存储不同类型的图像和数码数据的文件, 并且可以在不同类型的机器上传输, 同时还有统一处理这种文件格式的函数库。Caffe 训练模型可以选择保存为 HDF5 格式或 (默认的) ProtoBuffer 格式。

(1) 下载 hdf5-1.8.9.tar.gz 后解压:

```
tar zxvf hdf5-1.8.9.tar.gz
cd hdf5-1.8.9/
```

(2) 配置:

```
./configure --prefix=/home/yourname/local_install/
```

(3) 编译安装:

```
make -j && make install
```

HDF5 的使用方法可参考 Caffe 源码中的 `hdf5.hpp` 和 `hdf5.cpp`。

5.7 OpenCV

OpenCV 是世界上最流行的开源计算机视觉库，包含有大量图像处理函数。Caffe 使用 OpenCV 完成一些图像存取和预处理功能。

下载 `opencv-3.0.0.zip` 并解压:

```
unzip opencv-3.0.0.zip
cd opencv-3.0.0/
mkdir build;
cd build/
cmake ..
ccmake ..
```

很多同学在编译 OpenCV 时很是头大，因为配置选项较多，编译时总是报各种莫名其妙的错误。其实 Caffe 里面用到的 OpenCV 模块非常有限，仅限于图片读写、图片缩放等 CPU 上的模块，完全可以禁用无关模块以节省编译时间。

```
make && make install
```

OpenCV 的使用方法可以参考 Caffe 源码中的 `io.hpp` 和 `io.cpp`。

5.8 LMDB 和 LEVELDB

LMDB (Lightning Memory-Mapped Database Manager) ——闪电般的内存映射型数据库管理器，在 Caffe 中的作用主要是提供数据管理，将形形色色的原始数据 (JPEG 图片、二进制数据) 转换为统一的 Key-Value 存储，便于 Caffe 的 `DataLayer` 获取这些数据。LEVELDB 库是 Caffe 早期版本使用的数据存储方式，由 Google 开发。它是一种持续的键值对存储方式，键和值可以为任意字节数组。键的存储顺序可由用户定义的比较函数决定。目前大部分例程都已经使用

LMDB 代替了 LEVELDB，但是为了与以前的版本兼容，仍然将这个依赖库编译到 Caffe 中。

下载 LMDB 源码。无须配置，直接编译：

```
$ make
```

编译成功后，将 `lmdb.h` 拷贝到 `/home/yourname/local_install/include`，将编译生成的 `liblmdb.so` 拷贝到 `/home/yourname/local_install/lib/`

下载 `leveldb-1.7.0.tar.gz`，解压：

```
tar zxvf leveldb-1.7.0.tar.gz *
make
cp -r include/leveldb ~/local_install/include/
cp libleveldb.so* ~/local_install/lib/
```

LMDB 和 LEVELDB 的使用方法可以参考 Caffe 源码中的 `db_lmdb.hpp`、`db_lmdb.cpp`、`db_leveladb.hpp` 和 `db_leveladb.cpp`。

5.9 Snappy

Snappy 是一个用来压缩和解压缩的 C++ 库，旨在提供较高的压缩速度和合理的压缩率。Snappy 比 `zlib` 更快，但文件相对要大 20%~100%。

(1) 下载 `snappy-1.1.1.tar.gz` 并解压：

```
tar zxvf snappy-1.1.1.tar.gz
cd snappy-1.1.1/
```

(2) 配置：

```
./configure --prefix=/home/yourname/local_install/
```

(3) 编译安装：

```
make && make install
```

5.10 小结

按照前几节介绍编译、安装所有依赖包后，我们检查一下今天的工作收获：

```
$ cd /home/yourname/local_install/
$ tree -d -L 2
.
├── bin          // 存放编译生成的可执行程序
├── include      // 存放依赖包 API 的头文件，编译时加-I 包含该目录
│   ├── boost
│   ├── gflags
│   ├── glog
│   ├── google
│   ├── leveldb
│   ├── opencv
│   └── opencv2
├── lib          // 存放依赖包编译后生成的库文件，编译时加-L 包含该目录
│   ├── cmake  // 运行时记得将该目录加入 LD_LIBRARY_PATH 环境变量中
│   ├── pkgconfig
│   └── python2.7
├── share        // 额外文件
├── doc
├── hdf5_examples
└── OpenCV

17 directories
```

将依赖从系统安装包切换到今天手动编译的依赖包，需要修改 Caffe 目录下的 Makefile.config，找到如下几行：

```
# 额外头文件、库包含选项，我们需要添加今天的所有依赖包安装路径
INCLUDE_DIRS := /home/yourname/local_install/include $(PYTHON_INCLUDE) \
/usr/local/include
LIBRARY_DIRS :=/home/yourname/local_install/lib $(PYTHON_LIB) \
/usr/local/lib /usr/lib
# 注意将今天的依赖包路径放在系统路径前面，可保证先引用的是编译包而不是系统包
```

在 Caffe 根目录下执行 make，看看有什么反应？

① 如果很不幸的话，会看到如下报错信息：

```
$ make
PROTOC src/caffe/proto/caffe.proto
make: protoc: Command not found
make: *** [.build_release/src/caffe/proto/caffe.pb.h] Error 127
```

说明没找到 `protoc` 命令，此时应参照 5.1 节安装 ProtoBuffer 工具。如果安装了仍然报错，则需要参照 5.1 节的步骤设置 `PATH` 环境变量。

② 如果编译 Caffe 时报错信息为：

```
./include/caffe/common.hpp:4:32: fatal error: boost/shared_ptr.hpp: No such file or
directory
#include <boost/shared_ptr.hpp>
^
compilation terminated.
make: *** [.build_release/src/caffe/layer_factory.o] Error 1
```

说明没有安装 Boost 库，应参照 5.2 节安装 Boost。

③ 继续 make。

如果 make 报错信息如下：

```
./include/caffe/common.hpp:5:27: fatal error: gflags/gflags.h: No such file or directory
#include <gflags/gflags.h>
^
compilation terminated.
make: *** [.build_release/src/caffe/layer_factory.o] Error 1
```

说明没有安装 GFLAGS，请参考 5.3 节步骤安装 GFLAGS。

④ 再次执行 make 编译 Caffe，报错信息如下：

```
$ make
PROTOC src/caffe/proto/caffe.proto
CXX .build_release/src/caffe/proto/caffe.pb.cc
CXX src/caffe/layer_factory.cpp
In file included from ./include/caffe/blob.hpp:8:0,
                 from ./include/caffe/layer.hpp:8,
                 from src/caffe/layer_factory.cpp:8:
./include/caffe/common.hpp:6:26: fatal error: glog/logging.h: No such file or directory
#include <glog/logging.h>
^
compilation terminated.
make: *** [.build_release/src/caffe/layer_factory.o] Error 1
```

需按照 5.4 节步骤安装 GLOG。

⑤ 如果编译 Caffe 时报错信息为：

```
$ make
CXX src/caffe/layer_factory.cpp
In file included from ./include/caffe/common.hpp:19:0,
    from ./include/caffe/blob.hpp:8,
    from ./include/caffe/layer.hpp:8,
    from src/caffe/layer_factory.cpp:8:
./include/caffe/util/device_alternate.hpp:34:23: fatal error: cublas_v2.h: No such file
or directory
#include <cublas_v2.h>
    ^
compilation terminated.
make: *** [.build_release/src/caffe/layer_factory.o] Error 1
```

则报错原因是没有找到 cuBLAS 头文件。解决方法有两种：一是如果没有 GPU 硬件设备，需修改 Makefile.config，打开 CPU_ONLY 选项即可，编译时会自动绕过 CUDA 相关的源码，这样只能在 CPU 上运行 Caffe；二是可参考后面第 15 天内容安装 CUDA、cuDNN，自动消除该错误。

⑥ 如果编译 Caffe 时报错信息为：

```
$ make
CXX src/caffe/layer_factory.cpp
In file included from ./include/caffe/util/math_functions.hpp:11:0,
    from ./include/caffe/syncedmem.hpp:7,
    from ./include/caffe/blob.hpp:10,
    from ./include/caffe/layer.hpp:8,
    from src/caffe/layer_factory.cpp:8:
./include/caffe/util/mkl_alternate.hpp:11:19: fatal error: cblas.h: No such file or
directory
#include <cblas.h>
    ^
compilation terminated.
make: *** [.build_release/src/caffe/layer_factory.o] Error 1
```

解决方法须参考 5.5 节的 OpenBLAS 安装过程。

⑦ 如编译 Caffe 时，报错信息如下：

```
$ make
CXX src/caffe/layer_factory.cpp
In file included from ./include/caffe/common_layers.hpp:10:0,
```

```

    from ./include/caffe/vision_layers.hpp:10,
    from src/caffe/layer_factory.cpp:11:
./include/caffe/data_layers.hpp:8:18: fatal error: hdf5.h: No such file or directory
#include "hdf5.h"
^
compilation terminated.
make: *** [.build_release/src/caffe/layer_factory.o] Error 1

```

则参考 5.6 节的 HDF5 安装过程来加以解决。

⑧ 如编译 Caffe 时报错信息如下：

```

$ make
CXX src/caffe/layer_factory.cpp
CXX src/caffe/util/io.cpp
src/caffe/util/io.cpp:5:33: fatal error: opencv2/core/core.hpp: No such file or directory
#include <opencv2/core/core.hpp>
^
compilation terminated.
make: *** [.build_release/src/caffe/util/io.o] Error 1

```

说明未安装 OpenCV，参考 5.7 节进行安装。

⑨ 安装 OpenCV 3.0 后，如果 Caffe make 时仍报错：

```

CXX/LD -o .build_release/tools/convert_imageset.bin
.build_release/lib/libcaffe.so: undefined reference to cv::imread(cv::String const&,
int)'.build_release/lib/libcaffe.so: undefined reference to cv::imencode(cv::String
const&, cv::_InputArray const&, std::vector >&, std::vector > const&)'

```

原因就是 OpenCV 3.0 把 imread 相关函数放到 imgcodecs.lib 中了，而非原来的 imgproc.lib 中。

解决方法为修改 Makefile 文件（注意不是 Makefile.config），在

```

LIBRARIES += glog gflags protobuf leveldb snappy \
lmdb boost_system hdf5_hl hdf5 m \
opencv_core opencv_highgui opencv_imgproc opencv_imgcodecs

```

位置的最后添加 opencv_imgcodecs 即可。新版 Caffe 通过在 Makefile.config 中增加编译选项（OPENCV_VERSION := 3）修复了这一问题。

⑩ 没有安装 LMDB 时，编译 Caffe 会有如下报错信息：

```
$ make
```

```

CXX src/caffe/util/io.cpp
CXX src/caffe/util/cudnn.cpp
CXX src/caffe/util/db_lmdb.cpp
In file included from src/caffe/util/db_lmdb.cpp:1:0:
./include/caffe/util/db_lmdb.hpp:6:18: fatal error: lmdb.h: No such file or directory
#include "lmdb.h"
^
compilation terminated.
make: *** [.build_release/src/caffe/util/db_lmdb.o] Error 1

```

解决方法请参考 5.8 节的 LMDB 编译过程。

⑪ 未安装 LEVELDB 时，编译 Caffe 报错信息如下：

```

$ make
CXX src/caffe/util/db_lmdb.cpp
CXX src/caffe/util/im2col.cpp
CXX src/caffe/util/insert_splits.cpp
CXX src/caffe/util/upgrade_proto.cpp
CXX src/caffe/util/db.cpp
In file included from src/caffe/util/db.cpp:2:0:
./include/caffe/util/db_leveldb.hpp:6:24: fatal error: leveldb/db.h: No such file or
directory
#include "leveldb/db.h"
^
compilation terminated.
make: *** [.build_release/src/caffe/util/db.o] Error 1

```

需要参考 5.8 节安装 LEVELDB。

⑫ 编译 Caffe 时报错信息如下：

```

AR -o .build_release/lib/libcaffe.a
LD -o .build_release/lib/libcaffe.so
/usr/bin/ld: cannot find -lsnappy
collect2: error: ld returned 1 exit status
make: *** [.build_release/lib/libcaffe.so] Error 1

```

表示缺少 Snappy 库，参考 5.9 节安装即可。

到此为止，Caffe 编译一切正常。

继续编译、运行测试代码：

```
$ export LD_LIBRARY_PATH=/home/yourname/local_install/lib:$LD_LIBRARY_PATH
$ export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
$ make test
$ make runtest
```

如果希望使用 Python 或 Matlab 调用 Caffe，则还需编译外壳：

```
$ make pycaffe
$ make matcaffe
```

最后还可编译生成可发布安装包，用来交付给其他调用方。

```
$ make distribute
```

掌握了今天和昨天的内容，并能举一反三地解决其他开源项目安装过程中的问题，会让你对各类大型软件系统部署更加得心应手，可成长为优秀的系统工程师。

5.11 练习题

思考：为什么没有 igemm 函数？

5.12 参考资料

[1] 百度云盘 (<http://pan.baidu.com/s/1sks4XFv>, 提取码: idi7)

第 6 天

运行手写体数字识别例程

今天我们将通过运行一个完整的例子来熟悉 Caffe 的基本使用。

6.1 MNIST 数据集

MNIST (Mixed National Institute of Standards and Technology) 是一个大型的手写体数字数据库, 广泛用于机器学习领域的训练和测试, 由纽约大学 Yann LeCun 教授整理, 下载链接见参考资料[1]。MNIST 包括 60000 个训练集和 10000 个测试集, 每张图都已经进行尺寸归一化、数字居中处理, 固定尺寸为 28 像素 \times 28 像素。图 6-1 展示了 MNIST 中一些样本。



图6-1 MNIST样本

6.1.1 下载 MNIST 数据集

MNIST 数据集可以在 Caffe 源码框架的 data/mnist 下用 get_mnist.sh 脚本下载。

```
$ cd data/mnist/  
$ ./get_mnist.sh
```

```
$ tree
.
├── get_mnist.sh
├── t10k-images-idx3-ubyte
├── t10k-labels-idx1-ubyte
├── train-images-idx3-ubyte
└── train-labels-idx1-ubyte
```

0 directories, 5 files

我们看一下 `get_mnist.sh` 脚本内容。

```
#!/usr/bin/env sh
# 该脚本用于下载 MNIST 数据集并解压

DIR="$( cd "$(dirname "$0")" ; pwd -P )"
cd $DIR

echo "Downloading..."

wget --no-check-certificate http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
wget --no-check-certificate http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
wget --no-check-certificate http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
wget --no-check-certificate http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz

echo "Unzipping..."

gunzip train-images-idx3-ubyte.gz
gunzip train-labels-idx1-ubyte.gz
gunzip t10k-images-idx3-ubyte.gz
gunzip t10k-labels-idx1-ubyte.gz

# Creation is split out because leveldb sometimes causes segfault
# and needs to be re-created.

echo "Done."
```

可见，MNIST 原始数据为 4 个文件，如表 6-1 所示。

6.1.2 MNIST 数据格式描述

MNIST 具体的文件格式描述如表 6-2 至表 6-5 所示。

表 6-1 MNIST 原始数据文件

文 件 名	说 明
train-images-idx3-ubyte	训练集，图片
train-labels-idx1-ubyte	训练集，标签
t10k-images-idx3-ubyte	测试集，图片
t10k-labels-idx1-ubyte	测试集，标签

表 6-2 训练集图片文件格式描述

train-images-idx3-ubyte			
文件偏移量	数据类型	值	描 述
0000	32 位整型	2051	魔数（大端存储）
0004	32 位整型	60000	文件包含条目总数
0008	32 位整型	28	行数
0012	32 位整型	28	列数
0016	8 位字节	?	像素值（0~255）
0017	8 位字节	?	像素值（0~255）
...

表 6-3 训练集标签文件格式描述

train-labels-idx1-ubyte			
文件偏移量	数据类型	值	描 述
0000	32 位整型	2049	魔数（大端存储）
0004	32 位整型	60000	文件包含条目总数
0008	8 位字节	?	标签值（0~9）
0009	8 位字节	?	标签值（0~9）
...

表 6-4 测试集图片文件格式描述

t10k-images-idx3-ubyte			
文件偏移量	数据类型	值	描 述
0000	32 位整型	2051	魔数（大端存储）
0004	32 位整型	10000	文件包含条目总数
0008	32 位整型	28	行数
0012	32 位整型	28	列数
0016	8 位字节	?	像素值（0~255）
0017	8 位字节	?	像素值（0~255）
...

表 6-5 测试集标签文件格式描述

t10k-labels-idx1-ubyte			
文件偏移量	数据类型	值	描 述
0000	32 位整型	2049	魔数（大端存储）
0004	32 位整型	10000	文件包含条目总数
0008	8 位字节	?	标签值（0~9）
0009	8 位字节	?	标签值（0~9）
...

注意：图片文件中像素按行组织，像素值 0 表示背景（白色），像素值 255 表示前景（黑色）。

6.1.3 转换格式

下载到的原始数据集为二进制文件，需要转换为 LEVELDB 或 LMDB 才能被 Caffe 识别。我们已经编译好 Caffe，只需在 Caffe 根目录下执行以下脚本：

```
$ ./examples/mnist/create_mnist.sh
Creating lmdb...
Done.
```

浏览例程所在目录 examples/mnist，发现生成了 examples/mnist/mnist_train_lmdb/ 和 examples/mnist/mnist_test_lmdb/ 两个目录，每个目录下都有两个文件：data.mdb 和 lock.mdb。

```
$ ls -l examples/mnist/mnist_train_lmdb/
total 60320
-rw-rw-r-- 1 yourname yourname 61763584 Oct 16 01:45 data.mdb
-rw-rw-r-- 1 yourname yourname 8192 Oct 16 01:45 lock.mdb
$ ls -l examples/mnist/mnist_test_lmdb/
total 10100
-rw-rw-r-- 1 yourname yourname 10338304 Oct 16 01:45 data.mdb
-rw-rw-r-- 1 yourname yourname 8192 Oct 16 01:45 lock.mdb
```

顾名思义，mnist_train_lmdb 就是生成的 LMDB 格式的 MNIST 训练集，mnist_test_lmdb 则为 LMDB 格式的测试集（又称为验证集）。

```
#!/usr/bin/env sh
# 该脚本将 MNIST 原始数据转换为 lmdb/leveldb 格式

# lmdb/leveldb 生成路径
```

```

EXAMPLE=examples/mnist
# 原始数据路径
DATA=data/mnist
# 二进制文件路径
BUILD=build/examples/mnist

# 后端类型，可选 lmdb/leveldb
BACKEND="lmdb"

echo "Creating ${BACKEND}..."

# 如果已经存在 lmdb/leveldb，则先删除
rm -rf $EXAMPLE/mnist_train_${BACKEND}
rm -rf $EXAMPLE/mnist_test_${BACKEND}

# 创建训练集 db
$BUILD/convert_mnist_data.bin $DATA/train-images-idx3-ubyte \
    $DATA/train-labels-idx1-ubyte $EXAMPLE/mnist_train_${BACKEND} --backend=${BACKEND}

# 创建测试集 db
$BUILD/convert_mnist_data.bin $DATA/t10k-images-idx3-ubyte \
    $DATA/t10k-labels-idx1-ubyte $EXAMPLE/mnist_test_${BACKEND} --backend=${BACKEND}

echo "Done."

```

上述脚本调用了 `build/examples/mnist/convert_mnist_data.bin` 这个可执行程序，对应的源文件为 `examples/mnist/convert_mnist_data.cpp`，用 `vi` 打开这个源文件看一下：

```

// 本程序将 MNIST 数据集转换为（默认的）lmdb 或
// leveldb（--backend=leveldb）格式，便于 Caffe 载入数据
// 命令行参数说明：
//   convert_mnist_data [FLAGS] 输入图片文件、输入标签文件、输出 db 文件

#include <gflags/gflags.h>
#include <glog/logging.h>
#include <google/protobuf/text_format.h>
#include <leveldb/db.h>
#include <leveldb/write_batch.h>
#include <lmdb.h>
#include <stdint.h>

```

```

#include <sys/stat.h>

#include <fstream> // NOLINT(readability/streams)
#include <string>

#include "caffe/proto/caffe.pb.h"

using namespace caffe; // NOLINT(build/namespaces)
using std::string;

// GFLAGS 工具定义命令行选项 backend, 默认值为 lmdb, 即--backend=lmdb
DEFINE_string(backend, "lmdb", "The backend for storing the result");

// 大小端转换。MNIST 原始数据文件中 32 位整型值为大端存储, C/C++ 变量为小端存储, 因此需要加入转换机制
uint32_t swap_endian(uint32_t val) {
    val = ((val << 8) & 0xFF00FF00) | ((val >> 8) & 0xFF00FF);
    return (val << 16) | (val >> 16);
}

void convert_dataset(const char* image_filename, const char* label_filename,
    const char* db_path, const string& db_backend) {
    // 用 C++ 输入文件流以二进制方式打开文件
    std::ifstream image_file(image_filename, std::ios::in | std::ios::binary);
    std::ifstream label_file(label_filename, std::ios::in | std::ios::binary);
    CHECK(image_file) << "Unable to open file " << image_filename;
    CHECK(label_file) << "Unable to open file " << label_filename;
    // 读取魔数和基本信息
    uint32_t magic;
    uint32_t num_items;
    uint32_t num_labels;
    uint32_t rows;
    uint32_t cols;

    // 读取魔数 (4 Bytes)
    image_file.read(reinterpret_cast<char*>(&magic), 4);
    // 大端到小端转换
    magic = swap_endian(magic);
    // 校验魔数是否为 2051, 不是则报错
    CHECK_EQ(magic, 2051) << "Incorrect image file magic.";
}

```

```

// 同理，校验标签文件
label_file.read(reinterpret_cast<char*>(&magic), 4);
magic = swap_endian(magic);
CHECK_EQ(magic, 2049) << "Incorrect label file magic.";
image_file.read(reinterpret_cast<char*>(&num_items), 4);
num_items = swap_endian(num_items);
label_file.read(reinterpret_cast<char*>(&num_labels), 4);
num_labels = swap_endian(num_labels);
CHECK_EQ(num_items, num_labels);
image_file.read(reinterpret_cast<char*>(&rows), 4);
rows = swap_endian(rows);
image_file.read(reinterpret_cast<char*>(&cols), 4);
cols = swap_endian(cols);

// lmdb 相关句柄
MDB_env *mdb_env;
MDB_dbi mdb_dbi;
MDB_val mdb_key, mdb_data;
MDB_txn *mdb_txn;
// leveldb 相关句柄
leveldb::DB* db;
leveldb::Options options;
options.error_if_exists = true;
options.create_if_missing = true;
options.write_buffer_size = 268435456;
leveldb::WriteBatch* batch = NULL;

// 打开 db
if (db_backend == "leveldb") { // leveldb
    LOG(INFO) << "Opening leveldb " << db_path;
    leveldb::Status status = leveldb::DB::Open(
        options, db_path, &db);
    CHECK(status.ok()) << "Failed to open leveldb " << db_path
    << ". Is it already existing?";
    batch = new leveldb::WriteBatch();
} else if (db_backend == "lmdb") { // lmdb
    LOG(INFO) << "Opening lmdb " << db_path;
    CHECK_EQ(mkdir(db_path, 0744), 0)
    << "mkdir " << db_path << "failed";

```



```

CHECK_EQ(mdb_env_create(&mdb_env), MDB_SUCCESS) << "mdb_env_create failed";
CHECK_EQ(mdb_env_set_mapsize(mdb_env, 1099511627776), MDB_SUCCESS) // 1TB
<< "mdb_env_set_mapsize failed";
CHECK_EQ(mdb_env_open(mdb_env, db_path, 0, 0664), MDB_SUCCESS)
<< "mdb_env_open failed";
CHECK_EQ(mdb_txn_begin(mdb_env, NULL, 0, &mdb_txn), MDB_SUCCESS)
<< "mdb_txn_begin failed";
CHECK_EQ(mdb_open(mdb_txn, NULL, 0, &mdb_dbi), MDB_SUCCESS)
<< "mdb_open failed. Does the lmbd already exist? ";
} else {
    LOG(FATAL) << "Unknown db backend " << db_backend;
}

// 将读取数据保存至db
char label;
char* pixels = new char[rows * cols];
int count = 0;
const int kMaxKeyLength = 10;
char key_cstr[kMaxKeyLength];
string value;

Datum datum;
datum.set_channels(1);
datum.set_height(rows);
datum.set_width(cols);
LOG(INFO) << "A total of " << num_items << " items.";
LOG(INFO) << "Rows: " << rows << " Cols: " << cols;
for (int item_id = 0; item_id < num_items; ++item_id) {
    image_file.read(pixels, rows * cols);
    label_file.read(&label, 1);
    datum.set_data(pixels, rows*cols);
    datum.set_label(label);
    snprintf(key_cstr, kMaxKeyLength, "%08d", item_id);
    datum.SerializeToString(&value);
    string keystr(key_cstr);

    // 放到数据库中
    if (db_backend == "leveldb") { // leveldb
        batch->Put(keystr, value);
    }
}

```

```

    } else if (db_backend == "lmdb") { // lmdb
        mdb_data.mv_size = value.size();
        mdb_data.mv_data = reinterpret_cast<void*>(&value[0]);
        mdb_key.mv_size = keystr.size();
        mdb_key.mv_data = reinterpret_cast<void*>(&keystr[0]);
        CHECK_EQ(mdb_put(mdb_txn, mdb_dbi, &mdb_key, &mdb_data, 0), MDB_SUCCESS)
    }
    << "mdb_put failed";

    } else {
        LOG(FATAL) << "Unknown db backend " << db_backend;
    }

    if (++count % 1000 == 0) {
        // 批量提交更改
        if (db_backend == "leveldb") { // leveldb
            db->Write(leveldb::WriteOptions(), batch);
            delete batch;
            batch = new leveldb::WriteBatch();
        } else if (db_backend == "lmdb") { // lmdb
            CHECK_EQ(mdb_txn_commit(mdb_txn), MDB_SUCCESS)
        }
        << "mdb_txn_commit failed";

        CHECK_EQ(mdb_txn_begin(mdb_env, NULL, 0, &mdb_txn), MDB_SUCCESS)
        << "mdb_txn_begin failed";

        } else {
            LOG(FATAL) << "Unknown db backend " << db_backend;
        }
    }

    // 写最后一个 batch
    if (count % 1000 != 0) {
        if (db_backend == "leveldb") { // leveldb
            db->Write(leveldb::WriteOptions(), batch);
            delete batch;
            delete db;
        } else if (db_backend == "lmdb") { // lmdb
            CHECK_EQ(mdb_txn_commit(mdb_txn), MDB_SUCCESS) << "mdb_txn_commit failed";
            mdb_close(mdb_env, mdb_dbi);
            mdb_env_close(mdb_env);
        } else {
            LOG(FATAL) << "Unknown db backend " << db_backend;
        }
    }
}

```

```

    }
    LOG(ERROR) << "Processed " << count << " files.";
}
delete[] pixels;
}

int main(int argc, char** argv) {
#ifdef GFLAGS_GFLAGS_H_
    namespace gflags = google;
#endif

    gflags::SetUsageMessage("This script converts the MNIST dataset to\n"
        "the lmdb/leveldb format used by Caffe to load data.\n"
        "Usage:\n"
        "    convert_mnist_data [FLAGS] input_image_file input_label_file "
        "output_db_file\n"
        "The MNIST dataset could be downloaded at\n"
        "    http://yann.lecun.com/exdb/mnist/\n"
        "You should gunzip them after downloading, "
        "or directly use data/mnist/get_mnist.sh\n");
    gflags::ParseCommandLineFlags(&argc, &argv, true);

    // FLAGS_backend 在前面通过 DEFINE_string 定义，是字符串类型
    const string& db_backend = FLAGS_backend;

    if (argc != 4) {
        gflags::ShowUsageWithFlagsRestrict(argv[0],
            "examples/mnist/convert_mnist_data");
    } else {
        google::InitGoogleLogging(argv[0]);
        convert_dataset(argv[1], argv[2], argv[3], db_backend);
    }
    return 0;
}

```

通过上述源代码，读者可以熟悉一下 LEVELDB 或 LMDB 的基本使用方法。在 Caffe 源码中还会再见到它们。

TIPS: Caffe 为什么采用 LMDB、LEVELDB，而不是直接读取原始数据？

答：一方面，数据类型多种多样（有二进制文件、文本文件、编码后的图像文件如 JPEG

或 PNG、网络爬取的数据等)，不可能用一套代码实现所有类型的输入数据读取，转换为统一格式可以简化数据读取层的实现；另一方面，使用 LMDB、LEVELDB 可以提高磁盘 IO 利用率。

6.2 LeNet-5 模型

接下来我们继续认识一个经典的深度学习模型——LeNet-5^[2]。该模型是 Yann LeCun 最早提出的，并应用到邮政编码识别中。

6.2.1 LeNet-5 模型描述

本节例程中的 LeNet-5 模型与原版稍有不同（例如，将 Sigmoid 激活函数改为 ReLU），其描述文件为 examples/mnist/lenet_train_val.prototxt，内容如下：

```
name: "LeNet"                // 网络 (Net) 的名称为 LeNet
layer {                      // 定义一个层 (Layer)
  name: "mnist"              // 层的名称为 mnist
  type: "Data"              // 层的类型为数据层
  top: "data"               // 层的输出 blob 有两个: data 和 label
  top: "label"
  include {
    phase: TRAIN            // 该层参数只在训练阶段有效
  }
  transform_param {
    scale: 0.00390625       // 数据变换使用的数据缩放因子
  }
  data_param {              // 数据层参数
    source: "examples/mnist/mnist_train_lmdb" // LMDB 的路径
    batch_size: 64          // 批量数目，一次读取 64 张图
    backend: LMDB           // 数据格式为 LMDB
  }
}

layer { // 一个新数据层，名字也叫 mnist，输出 blob 也是 data 和 label，但是这里定义
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
```

```

    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}

layer { // 定义一个新的卷积层 conv1, 输入 blob 为 data, 输出 blob 为 conv1
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1 // 权值学习速率倍乘因子, 1 倍表示保持与全局参数一致
  }
  param {
    lr_mult: 2 // bias 学习速率倍乘因子, 是全局参数的 2 倍
  }
  convolution_param { // 卷积计算参数
    num_output: 20 // 输出 feature map 数目为 20
    kernel_size: 5 // 卷积核尺寸, 5 × 5
    stride: 1 // 卷积输出跳跃间隔, 1 表示连续输出, 无跳跃
    weight_filler { // 权值使用 xavier 填充器
      type: "xavier"
    }
    bias_filler { // bias 使用常数填充器, 默认为 0
      type: "constant"
    }
  }
}

layer { // 定义新的下采样层 pool1, 输入 blob 为 conv1, 输出 blob 为 pool1
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
}

```

```
pooling_param {      // 下采样参数
    pool: MAX          // 使用最大值下采样方法
    kernel_size: 2     // 下采样窗口尺寸 2 × 2
    stride: 2          // 下采样输出跳跃间隔 2 × 2
}
}
layer { // 新的卷积层, 和 conv1 类似
    name: "conv2"
    type: "Convolution"
    bottom: "pool1"
    top: "conv2"
    param {
        lr_mult: 1
    }
    param {
        lr_mult: 2
    }
    convolution_param {
        num_output: 50
        kernel_size: 5
        stride: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
}
layer { // 新的下采样层, 和 pool1 类似
    name: "pool2"
    type: "Pooling"
    bottom: "conv2"
    top: "pool2"
    pooling_param {
        pool: MAX
        kernel_size: 2
        stride: 2
    }
}
```

```
}  
layer { // 新的全连接层, 输入 blob 为 pool2, 输出 blob 为 ip1  
  name: "ip1"  
  type: "InnerProduct"  
  bottom: "pool2"  
  top: "ip1"  
  param {  
    lr_mult: 1  
  }  
  param {  
    lr_mult: 2  
  }  
  inner_product_param { // 全连接层参数  
    num_output: 500 // 该层输出元素个数为 500  
    weight_filler {  
      type: "xavier"  
    }  
    bias_filler {  
      type: "constant"  
    }  
  }  
}  
layer { // 新的非线性层, 用 ReLU 方法  
  name: "relu1"  
  type: "ReLU"  
  bottom: "ip1"  
  top: "ip1"  
}  
layer {  
  name: "ip2"  
  type: "InnerProduct"  
  bottom: "ip1"  
  top: "ip2"  
  param {  
    lr_mult: 1  
  }  
  param {  
    lr_mult: 2  
  }  
}
```

```

inner_product_param {
  num_output: 10
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
  }
}
}

layer { // 分类准确率层，只在 Testing 阶段有效，输入 blob 为 ip2 和 label，输出 blob 为 accuracy，
  // 该层用于计算分类准确率
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip2"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}

layer { // 损失层，损失函数采用 SoftmaxLoss，输入 blob 为 ip2 和 label，输出 blob 为 loss
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}

```

该模型利用可视化工具绘制其结构，如图 6-2 所示。

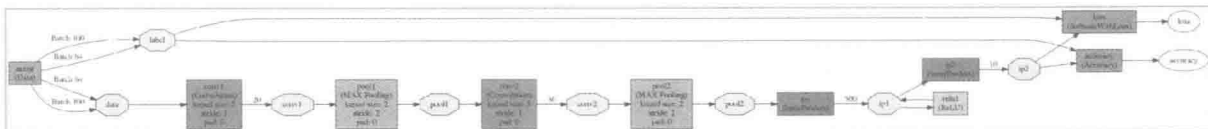


图6-2 LeNet模型结构图

通过 prototxt 和图 6-2 可以清晰地看到 LeNet 的网络结构。数据源 mnist 负责从预处理得到的 lmdb 数据库中读取图像数据 data 和标签数据 label，图像数据送入后续 CNN 结构中进行处理。CNN 结构包括一组由卷积层 conv(1,2)+下采样层 pool(1,2)交替形成的特征层，以及两个全连接

层 ip1 和 ip2（类似于多层感知器结构）。对 ip2 的输出进一步同标签数据 label 对比，可计算分类准确率 accuracy 和损失值 loss。LeNet 的设计蕴涵了 CNN 的精髓，理解该模型对设计更大模型（如 ImageNet 数据集上的 AlexNet、GoogleNet、VGG 等）有参考价值。

6.2.2 训练超参数

上面我们认识了 MNIST 手写体数字数据库和 LeNet CNN 模型，下面来正式启动 Caffe 训练过程，可以很快得到一个分类准确率在 99% 以上的模型。

运行 examples/mnist/train_lenet.sh 脚本。用 vi 打开该脚本，内容如下：

```
#!/usr/bin/env sh
./build/tools/caffe train --solver=examples/mnist/lenet_solver.prototxt
```

可见，调用了前面编译好的 build/tools/caffe.bin 二进制文件，参数--solver=examples/mnist/lenet_solver.prototxt 指定了训练超参数（Hyper-Parameter）文件，内容如下：

```
# 用于训练/预测的网络描述文件（ProtoBuffer 文本格式）
net: "examples/mnist/lenet_train_test.prototxt"
# 预测阶段迭代次数。在 MNIST 例程下，预测样本组（test batch）大小为 100
# 这里设置预测阶段迭代次数为 100 可以覆盖全部 10000 个测试集
test_iter: 100
# 训练时每迭代 500 次，进行一次预测
test_interval: 500
# 网络的基础学习速率、冲量和权衰量
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# 学习速率的衰减策略
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# 每经过 100 次迭代，在屏幕上打印一次运行 log
display: 100
# 最大迭代次数
max_iter: 10000
# 每 5000 次迭代打印一次快照
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
```

```
# Caffe 求解模式为 CPU 模式。有条件（见第 15 天内容）可以改为 GPU 模式
solver_mode: CPU
```

6.2.3 训练日志

基于 MNIST 数据集训练 LeNet 的运行日志如下（以下输出中的“//”后面内容为笔者注释，而非输出结果）：

```
$ ./examples/mnist/train_lenet.sh
// GLOG 输出格式：日期 时间 进程号 源码文件：代码行号] 输出信息
// 利用这些信息便于追踪源码运行流程，分析运行效率

// 使用 CPU 模式运行
I0404 15:16:14.372139 1965830144 caffe.cpp:178] Use CPU.
I0404 15:16:14.376407 1965830144 solver.cpp:48] Initializing solver from parameters:
// 打印训练超参数文件 examples/mnist/lenet_solver.prototxt 经过解析的内容
test_iter: 100
test_interval: 500
base_lr: 0.01
display: 100
max_iter: 10000
lr_policy: "inv"
gamma: 0.0001
power: 0.75
momentum: 0.9
weight_decay: 0.0005
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
solver_mode: CPU
net: "examples/mnist/lenet_train_test.prototxt" // 这里指定了 CNN 网络描述文件
I0404 15:16:14.380017 1965830144 solver.cpp:91] Creating training net from net file:
examples/mnist/lenet_train_test.prototxt
// 解析 CNN 网络描述文件中的网络参数，创建训练网络
I0404 15:16:14.387284 1965830144 net.cpp:313] The NetState phase (0) differed from the
phase (1) specified by a rule in layer mnist
I0404 15:16:14.387363 1965830144 net.cpp:313] The NetState phase (0) differed from the
phase (1) specified by a rule in layer accuracy
I0404 15:16:14.387398 1965830144 net.cpp:49] Initializing net from parameters:
// 打印训练网络参数描述
name: "LeNet"
```

```

state {
    phase: TRAIN
}

layer {
    name: "mnist"
    type: "Data"
    top: "data"
    top: "label"
    include {
        phase: TRAIN
    }
    transform_param {
        scale: 0.00390625
    }
    data_param {
        source: "examples/mnist/mnist_train_lmdb"
        batch_size: 64
        backend: LMDB
    }
}

//.....中间太长, 略

layer {
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "ip2"
    bottom: "label"
    top: "loss"
}

// 开始盖楼了! 先打地基 mnist
I0404 15:16:14.387723 1965830144 layer_factory.hpp:77] Creating layer mnist
I0404 15:16:14.397619 1965830144 net.cpp:91] Creating Layer mnist
// 产生两个输出, data 为图片数据, label 为标签数据
I0404 15:16:14.397647 1965830144 net.cpp:399] mnist -> data
I0404 15:16:14.397691 1965830144 net.cpp:399] mnist -> label
// 打开训练集 lmdb
I0404 15:16:14.401098 528384 db_lmdb.cpp:38] Opened lmdb examples/mnist/mnist_train_lmdb
// data 为四维数组, 又称 blob, 尺寸为 (64, 1, 28, 28)
I0404 15:16:14.402500 1965830144 data_layer.cpp:41] output data size: 64,1,28,28
I0404 15:16:14.403103 1965830144 net.cpp:141] Setting up mnist

```

```
I0404 15:16:14.403123 1965830144 net.cpp:148] Top shape: 64 1 28 28 (50176)
I0404 15:16:14.403136 1965830144 net.cpp:148] Top shape: 64 (64)
// 统计内存占用情况，会逐层累加
I0404 15:16:14.403144 1965830144 net.cpp:156] Memory required for data: 200960
// 盖1楼，conv1
I0404 15:16:14.403157 1965830144 layer_factory.hpp:77] Creating layer conv1
I0404 15:16:14.403182 1965830144 net.cpp:91] Creating Layer conv1
// conv1 需要一个输入 data（来自上一层 mnist），产生一个输出 conv1（送入下一层）
I0404 15:16:14.403250 1965830144 net.cpp:425] conv1 <- data
I0404 15:16:14.403270 1965830144 net.cpp:399] conv1 -> conv1
I0404 15:16:14.403623 1965830144 net.cpp:141] Setting up conv1
// conv1 输出的尺寸为（64，20，24，24）
I0404 15:16:14.403643 1965830144 net.cpp:148] Top shape: 64 20 24 24 (737280)
// 统计内存占用情况，逐层累加
I0404 15:16:14.403655 1965830144 net.cpp:156] Memory required for data: 3150080
I0404 15:16:14.403671 1965830144 layer_factory.hpp:77] Creating layer pool1
// .....中间层创建过程类似，不表
I0404 15:16:14.403687 1965830144 net.cpp:91] Creating Layer pool1
I0404 15:16:14.403743 1965830144 net.cpp:425] pool1 <- conv1
I0404 15:16:14.403756 1965830144 net.cpp:399] pool1 -> pool1
I0404 15:16:14.403983 1965830144 net.cpp:141] Setting up pool1
I0404 15:16:14.404001 1965830144 net.cpp:148] Top shape: 64 20 12 12 (184320)
I0404 15:16:14.404019 1965830144 net.cpp:156] Memory required for data: 3887360
I0404 15:16:14.404036 1965830144 layer_factory.hpp:77] Creating layer conv2
I0404 15:16:14.404063 1965830144 net.cpp:91] Creating Layer conv2
I0404 15:16:14.404078 1965830144 net.cpp:425] conv2 <- pool1
I0404 15:16:14.404098 1965830144 net.cpp:399] conv2 -> conv2
I0404 15:16:14.404726 1965830144 net.cpp:141] Setting up conv2
I0404 15:16:14.404755 1965830144 net.cpp:148] Top shape: 64 50 8 8 (204800)
I0404 15:16:14.404772 1965830144 net.cpp:156] Memory required for data: 4706560
I0404 15:16:14.404794 1965830144 layer_factory.hpp:77] Creating layer pool2
I0404 15:16:14.404824 1965830144 net.cpp:91] Creating Layer pool2
I0404 15:16:14.404834 1965830144 net.cpp:425] pool2 <- conv2
I0404 15:16:14.404845 1965830144 net.cpp:399] pool2 -> pool2
I0404 15:16:14.404862 1965830144 net.cpp:141] Setting up pool2
I0404 15:16:14.404871 1965830144 net.cpp:148] Top shape: 64 50 4 4 (51200)
I0404 15:16:14.404881 1965830144 net.cpp:156] Memory required for data: 4911360
I0404 15:16:14.404888 1965830144 layer_factory.hpp:77] Creating layer ip1
I0404 15:16:14.404901 1965830144 net.cpp:91] Creating Layer ip1
```

```
I0404 15:16:14.404911 1965830144 net.cpp:425] ip1 <- pool2
I0404 15:16:14.404925 1965830144 net.cpp:399] ip1 -> ip1
I0404 15:16:14.409618 1965830144 net.cpp:141] Setting up ip1
I0404 15:16:14.409646 1965830144 net.cpp:148] Top shape: 64 500 (32000)
I0404 15:16:14.409654 1965830144 net.cpp:156] Memory required for data: 5039360
I0404 15:16:14.409667 1965830144 layer_factory.hpp:77] Creating layer relu1
I0404 15:16:14.409683 1965830144 net.cpp:91] Creating Layer relu1
I0404 15:16:14.409692 1965830144 net.cpp:425] relu1 <- ip1
I0404 15:16:14.409699 1965830144 net.cpp:386] relu1 -> ip1 (in-place)
I0404 15:16:14.409709 1965830144 net.cpp:141] Setting up relu1
I0404 15:16:14.409715 1965830144 net.cpp:148] Top shape: 64 500 (32000)
I0404 15:16:14.409723 1965830144 net.cpp:156] Memory required for data: 5167360
I0404 15:16:14.409728 1965830144 layer_factory.hpp:77] Creating layer ip2
I0404 15:16:14.409741 1965830144 net.cpp:91] Creating Layer ip2
I0404 15:16:14.409747 1965830144 net.cpp:425] ip2 <- ip1
I0404 15:16:14.409755 1965830144 net.cpp:399] ip2 -> ip2
I0404 15:16:14.409821 1965830144 net.cpp:141] Setting up ip2
I0404 15:16:14.409827 1965830144 net.cpp:148] Top shape: 64 10 (640)
I0404 15:16:14.409834 1965830144 net.cpp:156] Memory required for data: 5169920
// 盖最后一层 loss
I0404 15:16:14.409842 1965830144 layer_factory.hpp:77] Creating layer loss
I0404 15:16:14.409853 1965830144 net.cpp:91] Creating Layer loss
// 该层需要两个输入 ip2 和 label, 产生一个输出 loss
I0404 15:16:14.409860 1965830144 net.cpp:425] loss <- ip2
I0404 15:16:14.409867 1965830144 net.cpp:425] loss <- label
I0404 15:16:14.409876 1965830144 net.cpp:399] loss -> loss
I0404 15:16:14.409898 1965830144 layer_factory.hpp:77] Creating layer loss
I0404 15:16:14.409919 1965830144 net.cpp:141] Setting up loss
// 输出 loss 尺寸为 1, loss weight 参数为 1
I0404 15:16:14.409925 1965830144 net.cpp:148] Top shape: (1)
I0404 15:16:14.409931 1965830144 net.cpp:151] with loss weight 1
// 统计内存占用情况, 需要 5169924B, 即 4.93MB
I0404 15:16:14.409942 1965830144 net.cpp:156] Memory required for data: 5169924
// 从后往前统计哪些层需要做反向传播计算 (BP)
I0404 15:16:14.409947 1965830144 net.cpp:217] loss needs backward computation.
I0404 15:16:14.409955 1965830144 net.cpp:217] ip2 needs backward computation.
I0404 15:16:14.409960 1965830144 net.cpp:217] relu1 needs backward computation.
I0404 15:16:14.409965 1965830144 net.cpp:217] ip1 needs backward computation.
I0404 15:16:14.409971 1965830144 net.cpp:217] pool2 needs backward computation.
```

```

I0404 15:16:14.409977 1965830144 net.cpp:217] conv2 needs backward computation.
I0404 15:16:14.409982 1965830144 net.cpp:217] pool1 needs backward computation.
I0404 15:16:14.409988 1965830144 net.cpp:217] conv1 needs backward computation.
I0404 15:16:14.409994 1965830144 net.cpp:219] mnist does not need backward computation.
I0404 15:16:14.410023 1965830144 net.cpp:261] This network produces output loss
// 盖楼完毕
I0404 15:16:14.410037 1965830144 net.cpp:274] Network initialization done.
// 还需创建测试网络，再盖一次楼
I0404 15:16:14.410277 1965830144 solver.cpp:181] Creating test net (#0) specified by net
file: examples/mnist/lenet_train_test.prototxt
I0404 15:16:14.410307 1965830144 net.cpp:313] The NetState phase (1) differed from the
phase (0) specified by a rule in layer mnist
I0404 15:16:14.410322 1965830144 net.cpp:49] Initializing net from parameters:
// 略，类似于第一座楼情况
// 只是地基 mnist 改了一下 lmdb 源和输出尺寸，顶楼加了个 accuracy 阁楼
name: "LeNet"
state {
  phase: TEST
}
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
//.....中间重复，不表
layer {
  name: "accuracy"
  type: "Accuracy"

```

```
bottom: "ip2"
bottom: "label"
top: "accuracy"
include {
  phase: TEST
}
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}
```

// 具体盖楼过程与训练网络类似

```
I0404 15:16:14.410482 1965830144 layer_factory.hpp:77] Creating layer mnist
I0404 15:16:14.410609 1965830144 net.cpp:91] Creating Layer mnist
I0404 15:16:14.410631 1965830144 net.cpp:399] mnist -> data
I0404 15:16:14.410643 1965830144 net.cpp:399] mnist -> label
I0404 15:16:14.413457 1601536 db_lmdb.cpp:38] Opened lmdb examples/mnist/mnist_test_lmdb
I0404 15:16:14.414069 1965830144 data_layer.cpp:41] output data size: 100,1,28,28
I0404 15:16:14.415026 1965830144 net.cpp:141] Setting up mnist
I0404 15:16:14.415066 1965830144 net.cpp:148] Top shape: 100 1 28 28 (78400)
I0404 15:16:14.415151 1965830144 net.cpp:148] Top shape: 100 (100)
I0404 15:16:14.415164 1965830144 net.cpp:156] Memory required for data: 314000
I0404 15:16:14.415179 1965830144 layer_factory.hpp:77] Creating layer label_mnist_1_split
I0404 15:16:14.415208 1965830144 net.cpp:91] Creating Layer label_mnist_1_split
I0404 15:16:14.415218 1965830144 net.cpp:425] label_mnist_1_split <- label
I0404 15:16:14.415237 1965830144 net.cpp:399] label_mnist_1_split -> label_mnist_1_split_0
I0404 15:16:14.415253 1965830144 net.cpp:399] label_mnist_1_split -> label_mnist_1_split_1
I0404 15:16:14.415269 1965830144 net.cpp:141] Setting up label_mnist_1_split
I0404 15:16:14.415277 1965830144 net.cpp:148] Top shape: 100 (100)
I0404 15:16:14.415287 1965830144 net.cpp:148] Top shape: 100 (100)
I0404 15:16:14.415294 1965830144 net.cpp:156] Memory required for data: 314800
I0404 15:16:14.415302 1965830144 layer_factory.hpp:77] Creating layer conv1
I0404 15:16:14.415356 1965830144 net.cpp:91] Creating Layer conv1
I0404 15:16:14.415380 1965830144 net.cpp:425] conv1 <- data
I0404 15:16:14.415393 1965830144 net.cpp:399] conv1 -> conv1
```

```
I0404 15:16:14.415449 1965830144 net.cpp:141] Setting up conv1
I0404 15:16:14.415459 1965830144 net.cpp:148] Top shape: 100 20 24 24 (1152000)
I0404 15:16:14.415472 1965830144 net.cpp:156] Memory required for data: 4922800
I0404 15:16:14.415518 1965830144 layer_factory.hpp:77] Creating layer pool1
I0404 15:16:14.415539 1965830144 net.cpp:91] Creating Layer pool1
I0404 15:16:14.415549 1965830144 net.cpp:425] pool1 <- conv1
I0404 15:16:14.415560 1965830144 net.cpp:399] pool1 -> pool1
I0404 15:16:14.415583 1965830144 net.cpp:141] Setting up pool1
I0404 15:16:14.415592 1965830144 net.cpp:148] Top shape: 100 20 12 12 (288000)
I0404 15:16:14.415603 1965830144 net.cpp:156] Memory required for data: 6074800
I0404 15:16:14.415611 1965830144 layer_factory.hpp:77] Creating layer conv2
I0404 15:16:14.415627 1965830144 net.cpp:91] Creating Layer conv2
I0404 15:16:14.415637 1965830144 net.cpp:425] conv2 <- pool1
I0404 15:16:14.415647 1965830144 net.cpp:399] conv2 -> conv2
I0404 15:16:14.416070 1965830144 net.cpp:141] Setting up conv2
I0404 15:16:14.416085 1965830144 net.cpp:148] Top shape: 100 50 8 8 (320000)
I0404 15:16:14.416095 1965830144 net.cpp:156] Memory required for data: 7354800
I0404 15:16:14.416115 1965830144 layer_factory.hpp:77] Creating layer pool2
I0404 15:16:14.416127 1965830144 net.cpp:91] Creating Layer pool2
I0404 15:16:14.416136 1965830144 net.cpp:425] pool2 <- conv2
I0404 15:16:14.416193 1965830144 net.cpp:399] pool2 -> pool2
I0404 15:16:14.416224 1965830144 net.cpp:141] Setting up pool2
I0404 15:16:14.416232 1965830144 net.cpp:148] Top shape: 100 50 4 4 (80000)
I0404 15:16:14.416242 1965830144 net.cpp:156] Memory required for data: 7674800
I0404 15:16:14.416250 1965830144 layer_factory.hpp:77] Creating layer ip1
I0404 15:16:14.416262 1965830144 net.cpp:91] Creating Layer ip1
I0404 15:16:14.416270 1965830144 net.cpp:425] ip1 <- pool2
I0404 15:16:14.416285 1965830144 net.cpp:399] ip1 -> ip1
I0404 15:16:14.421391 1965830144 net.cpp:141] Setting up ip1
I0404 15:16:14.421423 1965830144 net.cpp:148] Top shape: 100 500 (50000)
I0404 15:16:14.421432 1965830144 net.cpp:156] Memory required for data: 7874800
I0404 15:16:14.421445 1965830144 layer_factory.hpp:77] Creating layer relul
I0404 15:16:14.421458 1965830144 net.cpp:91] Creating Layer relul
I0404 15:16:14.421465 1965830144 net.cpp:425] relul <- ip1
I0404 15:16:14.421478 1965830144 net.cpp:386] relul -> ip1 (in-place)
I0404 15:16:14.421489 1965830144 net.cpp:141] Setting up relul
I0404 15:16:14.421494 1965830144 net.cpp:148] Top shape: 100 500 (50000)
I0404 15:16:14.421500 1965830144 net.cpp:156] Memory required for data: 8074800
I0404 15:16:14.421506 1965830144 layer_factory.hpp:77] Creating layer ip2
```



```
I0404 15:16:14.421516 1965830144 net.cpp:91] Creating Layer ip2
I0404 15:16:14.421522 1965830144 net.cpp:425] ip2 <- ip1
I0404 15:16:14.421530 1965830144 net.cpp:399] ip2 -> ip2
I0404 15:16:14.421604 1965830144 net.cpp:141] Setting up ip2
I0404 15:16:14.421612 1965830144 net.cpp:148] Top shape: 100 10 (1000)
I0404 15:16:14.421617 1965830144 net.cpp:156] Memory required for data: 8078800
// 注意, ip2_ip2_0_split 在网络描述中没有显式给出, 是Caffe解析后自动加上的
I0404 15:16:14.421625 1965830144 layer_factory.hpp:77] Creating layer ip2_ip2_0_split
I0404 15:16:14.421635 1965830144 net.cpp:91] Creating Layer ip2_ip2_0_split
// ip2_ip2_0_split 接受一个输入 ip2
// 产生两个输出 ip2_ip2_0_split_0 和 ip2_ip2_0_split_1, 是复制关系
I0404 15:16:14.421641 1965830144 net.cpp:425] ip2_ip2_0_split <- ip2
I0404 15:16:14.421648 1965830144 net.cpp:399] ip2_ip2_0_split -> ip2_ip2_0_split_0
I0404 15:16:14.421658 1965830144 net.cpp:399] ip2_ip2_0_split -> ip2_ip2_0_split_1
I0404 15:16:14.421666 1965830144 net.cpp:141] Setting up ip2_ip2_0_split
I0404 15:16:14.421672 1965830144 net.cpp:148] Top shape: 100 10 (1000)
I0404 15:16:14.421679 1965830144 net.cpp:148] Top shape: 100 10 (1000)
I0404 15:16:14.421684 1965830144 net.cpp:156] Memory required for data: 8086800
// 想起了太极口诀: 一个大西瓜, 平均切两块, 一块儿给你, 一块儿给他……
// ip2_ip2_0_split_0 给了 accuracy 层
I0404 15:16:14.421689 1965830144 layer_factory.hpp:77] Creating layer accuracy
I0404 15:16:14.421712 1965830144 net.cpp:91] Creating Layer accuracy
I0404 15:16:14.421732 1965830144 net.cpp:425] accuracy <- ip2_ip2_0_split_0
I0404 15:16:14.421739 1965830144 net.cpp:425] accuracy <- label_mnist_1_split_0
I0404 15:16:14.421751 1965830144 net.cpp:399] accuracy -> accuracy
I0404 15:16:14.421761 1965830144 net.cpp:141] Setting up accuracy
// accuracy 层输出尺寸为1, 即分类准确率
I0404 15:16:14.421767 1965830144 net.cpp:148] Top shape: (1)
I0404 15:16:14.421773 1965830144 net.cpp:156] Memory required for data: 8086804
// ip2_ip2_0_split_1 给了 loss 层
I0404 15:16:14.421778 1965830144 layer_factory.hpp:77] Creating layer loss
I0404 15:16:14.421787 1965830144 net.cpp:91] Creating Layer loss
I0404 15:16:14.421792 1965830144 net.cpp:425] loss <- ip2_ip2_0_split_1
I0404 15:16:14.421798 1965830144 net.cpp:425] loss <- label_mnist_1_split_1
I0404 15:16:14.421805 1965830144 net.cpp:399] loss -> loss
I0404 15:16:14.421814 1965830144 layer_factory.hpp:77] Creating layer loss
I0404 15:16:14.421830 1965830144 net.cpp:141] Setting up loss
I0404 15:16:14.421838 1965830144 net.cpp:148] Top shape: (1)
I0404 15:16:14.421844 1965830144 net.cpp:151] with loss weight 1
```

```

I0404 15:16:14.421851 1965830144 net.cpp:156] Memory required for data: 8086808
I0404 15:16:14.421857 1965830144 net.cpp:217] loss needs backward computation.
I0404 15:16:14.421862 1965830144 net.cpp:219] accuracy does not need backward computation.
I0404 15:16:14.421869 1965830144 net.cpp:217] ip2_ip2_0_split needs backward computation.
I0404 15:16:14.421875 1965830144 net.cpp:217] ip2 needs backward computation.
I0404 15:16:14.421880 1965830144 net.cpp:217] relu1 needs backward computation.
I0404 15:16:14.421885 1965830144 net.cpp:217] ip1 needs backward computation.
I0404 15:16:14.421890 1965830144 net.cpp:217] pool2 needs backward computation.
I0404 15:16:14.421898 1965830144 net.cpp:217] conv2 needs backward computation.
I0404 15:16:14.421905 1965830144 net.cpp:217] pool1 needs backward computation.
I0404 15:16:14.421911 1965830144 net.cpp:217] conv1 needs backward computation.
I0404 15:16:14.421916 1965830144 net.cpp:219] label_mnist_1_split does not need backward
computation.
I0404 15:16:14.421923 1965830144 net.cpp:219] mnist does not need backward computation.
I0404 15:16:14.421928 1965830144 net.cpp:261] This network produces output accuracy
I0404 15:16:14.421934 1965830144 net.cpp:261] This network produces output loss
// 第二座楼盖好了
I0404 15:16:14.421944 1965830144 net.cpp:274] Network initialization done.
// 装修方案确定了
I0404 15:16:14.422015 1965830144 solver.cpp:60] Solver scaffolding done.
// 开始装修
I0404 15:16:14.422062 1965830144 caffe.cpp:219] Starting Optimization
I0404 15:16:14.422070 1965830144 solver.cpp:279] Solving LeNet
I0404 15:16:14.422075 1965830144 solver.cpp:280] Learning Rate Policy: inv
// 先测试一次，得到初始分类准确率和损失
I0404 15:16:14.423041 1965830144 solver.cpp:337] Iteration 0, Testing net (#0)
I0404 15:16:17.914536 1965830144 solver.cpp:404] Test net output #0: accuracy = 0.1014
I0404 15:16:17.914599 1965830144 solver.cpp:404] Test net output #1: loss = 2.33303
(* 1 = 2.33303 loss)
// 想都不用想，现在分类效果肯定很差，准确率只有 0.1 左右，和随机猜测准确率差不多……
// 损失值约为 2.3，等你读过第 13 天内容就知道这个数的来历了……
I0404 15:16:17.969997 1965830144 solver.cpp:228] Iteration 0, loss = 2.32217
// 训练网络在第 0 次迭代的分类性能，一样很差
// 注意，训练网络没有 accuracy 输出，只有 loss 输出
I0404 15:16:17.970052 1965830144 solver.cpp:244] Train net output #0: loss = 2.32217
(* 1 = 2.32217 loss)
I0404 15:16:17.970088 1965830144 sgd_solver.cpp:106] Iteration 0, lr = 0.01
I0404 15:16:22.859643 1965830144 solver.cpp:228] Iteration 100, loss = 0.238868
// 迭代 100 次的情况，loss 显著下降
I0404 15:16:22.859697 1965830144 solver.cpp:244] Train net output #0: loss = 0.238868

```

```
(* 1 = 0.238868 loss)
I0404 15:16:22.859709 1965830144 sgd_solver.cpp:106] Iteration 100, lr = 0.00992565
I0404 15:16:27.847350 1965830144 solver.cpp:228] Iteration 200, loss = 0.122524
I0404 15:16:27.847398 1965830144 solver.cpp:244] Train net output #0: loss = 0.122524
(* 1 = 0.122524 loss)
I0404 15:16:27.847410 1965830144 sgd_solver.cpp:106] Iteration 200, lr = 0.00985258
I0404 15:16:32.812690 1965830144 solver.cpp:228] Iteration 300, loss = 0.196796
I0404 15:16:32.812731 1965830144 solver.cpp:244] Train net output #0: loss = 0.196796
(* 1 = 0.196796 loss)
I0404 15:16:32.812743 1965830144 sgd_solver.cpp:106] Iteration 300, lr = 0.00978075
I0404 15:16:37.813115 1965830144 solver.cpp:228] Iteration 400, loss = 0.0882091
I0404 15:16:37.813197 1965830144 solver.cpp:244] Train net output #0: loss = 0.0882091
(* 1 = 0.0882091 loss)
I0404 15:16:37.813212 1965830144 sgd_solver.cpp:106] Iteration 400, lr = 0.00971013
// .....过了一段时间
// 每迭代 500 次, 做一次测试 (由 solver.prototxt 中超参数 test_interval: 500 设定)
I0404 15:16:42.784708 1965830144 solver.cpp:337] Iteration 500, Testing net (#0)
I0404 15:16:46.238106 1965830144 solver.cpp:404] Test net output #0: accuracy = 0.9722
// 不错哦, 准确率很快就达到 97%了
I0404 15:16:46.238154 1965830144 solver.cpp:404] Test net output #1: loss = 0.0920536
(* 1 = 0.0920536 loss)
// 继续训练.....中间输出重复, 略
I0404 15:20:56.985736 1965830144 solver.cpp:228] Iteration 4900, loss = 0.00676112
I0404 15:20:56.985781 1965830144 solver.cpp:244] Train net output #0: loss = 0.00676123
(* 1 = 0.00676123 loss)
I0404 15:20:56.985795 1965830144 sgd_solver.cpp:106] Iteration 4900, lr = 0.00741498
// 每迭代 5000 次, 打印一次快照 (由 solver.prototxt 中超参数 snapshot: 5000 设定)
I0404 15:21:02.056707 1965830144 solver.cpp:454] Snapshotting to binary proto file
examples/mnist/lenet_iter_5000.caffemodel
I0404 15:21:02.073658 1965830144 sgd_solver.cpp:273] Snapshotting solver state to binary
proto file examples/mnist/lenet_iter_5000.solverstate
I0404 15:21:02.083560 1965830144 solver.cpp:337] Iteration 5000, Testing net (#0)
I0404 15:21:05.723037 1965830144 solver.cpp:404] Test net output #0: accuracy = 0.9888
I0404 15:21:05.723078 1965830144 solver.cpp:404] Test net output #1: loss = 0.0349629
(* 1 = 0.0349629 loss)
// 继续训练.....中间内容重复, 略
// 最后一次打印快照
I0404 15:25:48.310060 1965830144 solver.cpp:454] Snapshotting to binary proto file
examples/mnist/lenet_iter_10000.caffemodel
I0404 15:25:48.324256 1965830144 sgd_solver.cpp:273] Snapshotting solver state to binary
proto file examples/mnist/lenet_iter_10000.solverstate
I0404 15:25:48.358948 1965830144 solver.cpp:317] Iteration 10000, loss = 0.0034473
I0404 15:25:48.358986 1965830144 solver.cpp:337] Iteration 10000, Testing net (#0)
I0404 15:25:51.754547 1965830144 solver.cpp:404] Test net output #0: accuracy = 0.9904
```

```
// 最终分类准确率为 99%
I0404 15:25:51.754601 1965830144 solver.cpp:404] Test net output #1: loss = 0.0321366
(* 1 = 0.0321366 loss)
// 最终 loss 值为 0.0321366
I0404 15:25:51.754614 1965830144 solver.cpp:322] Optimization Done.
I0404 15:25:51.754623 1965830144 caffe.cpp:222] Optimization Done.
// 装修结束，可以交付了
```

从上数输出结果可以看到最终训练的模型权值保存在 `examples/mnist/lenet_iter_10000.caffemodel` 文件中，训练状态保存在 `examples/mnist/lenet_iter_10000.solverstate` 文件中。这两个文件都是 ProtoBuffer 二进制格式文件（binary proto file）。

6.2.4 用训练好的模型对数据进行预测

利用训练好的 LeNet-5 模型权值文件（`examples/mnist/lenet_iter_10000.caffemodel`）可以对测试数据集（或外部测试集）进行预测。运行如下命令：

```
$ ./build/tools/caffe.bin test \
-model examples/mnist/lenet_train_test.prototxt \
-weights examples/mnist/lenet_iter_10000.caffemodel \
-iterations 100
```

命令行解释：

- `./build/tools/caffe.bin test`，表示只做预测（前向传播计算），不进行参数更新（后向传播计算）。
- `-model examples/mnist/lenet_train_test.prototxt`，指定模型描述文本文件。
- `-weights examples/mnist/lenet_iter_10000.caffemodel`，指定模型预先训练好的权值文件。
- `-iterations 100`，指定测试迭代次数。参与测试的样例数目为（`iterations * batch_size`），`batch_size` 在 model prototxt 中设定，为 100 时刚好覆盖全部 10000 个测试样本。

读者可以运行上述命令，分析输出日志，与训练阶段输出做对比。

6.2.5 Windows 下训练模型

Windows 用户你们还好吗？让我看到你们的双手！

在 Windows 下运行 Caffe 比较 trick，我们可以从 Linux 下配置好的 Caffe 环境中直接拷贝 LMDB 格式数据（`mnist_train_lmdb` 和 `mnist_test_lmdb`），读者可以从笔者的百度云盘^[3]获取，放入 `C:\Users\Administrator\Desktop\caffe-master\examples\mnist\`下。

打开 Windows 命令行, cd 到 C:\Users\Administrator\Desktop\caffe-master\, 运行命令如图 6-3 所示。

```
C:\Users\Administrator\Desktop\caffe-master>Build\x64\Release\caffe.exe train -s
solver_examples\mnist\lenet_solver.prototxt
```

图6-3 Windows Caffe 命令行参数

运行效果如图 6-4 所示。

```
管理员: C:\Windows\system32\cmd.exe - Build\x64\Release\caffe.exe tr...
10327 19:37:13.440347 1028 solver.cpp:244] Train net output #0: loss = 0.00
59689 (* 1 = 0.0059689 loss)
10327 19:37:13.440347 1028 sgd_solver.cpp:106] Iteration 7200, lr = 0.00665815
10327 19:37:13.633484 1028 solver.cpp:228] Iteration 7300, loss = 0.0218646
10327 19:37:13.633484 1028 solver.cpp:244] Train net output #0: loss = 0.02
18645 (* 1 = 0.0218645 loss)
10327 19:37:13.634485 1028 sgd_solver.cpp:106] Iteration 7400, lr = 0.00662927
10327 19:37:13.827622 1028 solver.cpp:228] Iteration 7500, loss = 0.00435396
10327 19:37:13.827622 1028 solver.cpp:244] Train net output #0: loss = 0.00
435387 (* 1 = 0.00435387 loss)
10327 19:37:13.827622 1028 sgd_solver.cpp:106] Iteration 7600, lr = 0.00660867
10327 19:37:14.018759 1028 solver.cpp:337] Iteration 7500, Testing net (#0)
10327 19:37:14.126834 1028 solver.cpp:404] Test net output #0: accuracy = 0
.9901
10327 19:37:14.126834 1028 solver.cpp:404] Test net output #1: loss = 0.071
681 (* 1 = 0.071681 loss)
10327 19:37:14.128836 1028 solver.cpp:228] Iteration 7500, loss = 0.00126071
10327 19:37:14.128836 1028 solver.cpp:244] Train net output #0: loss = 0.00
126063 (* 1 = 0.00126063 loss)
10327 19:37:14.128836 1028 sgd_solver.cpp:106] Iteration 7500, lr = 0.00657236
10327 19:37:14.328979 1028 solver.cpp:228] Iteration 7600, loss = 0.007851
10327 19:37:14.328979 1028 solver.cpp:244] Train net output #0: loss = 0.00
785092 (* 1 = 0.00785092 loss)
10327 19:37:14.329929 1028 sgd_solver.cpp:106] Iteration 7600, lr = 0.00654433
```

图6-4 Windows Caffe运行过程

运行结束时如图 6-5 所示。

```
管理员: C:\Windows\system32\cmd.exe
10327 19:37:18.913255 1028 solver.cpp:244] Train net output #0: loss = 0.00
300973 (* 1 = 0.00300973 loss)
10327 19:37:18.913255 1028 sgd_solver.cpp:106] Iteration 9700, lr = 0.00601302
10327 19:37:19.112373 1028 solver.cpp:228] Iteration 9800, loss = 0.0158959
10327 19:37:19.113374 1028 solver.cpp:244] Train net output #0: loss = 0.01
58959 (* 1 = 0.0158959 loss)
10327 19:37:19.113374 1028 sgd_solver.cpp:106] Iteration 9800, lr = 0.00599102
10327 19:37:19.312545 1028 solver.cpp:228] Iteration 9900, loss = 0.0045369
10327 19:37:19.313516 1028 solver.cpp:244] Train net output #0: loss = 0.00
453684 (* 1 = 0.00453684 loss)
10327 19:37:19.313516 1028 sgd_solver.cpp:106] Iteration 9900, lr = 0.00596843
10327 19:37:19.511656 1028 solver.cpp:454] Snapshotting to binary proto file ex
amples/mnist/lenet_iter_10000.caffemodel
10327 19:37:19.521663 1028 sgd_solver.cpp:273] Snapshotting solver state to bin
ary proto file examples/mnist/lenet_iter_10000.solverstate
10327 19:37:19.526667 1028 solver.cpp:317] Iteration 10000, loss = 0.00288106
10327 19:37:19.526667 1028 solver.cpp:337] Iteration 10000, Testing net (#0)
10327 19:37:19.633744 1028 solver.cpp:404] Test net output #0: accuracy = 0
.9914
10327 19:37:19.633744 1028 solver.cpp:404] Test net output #1: loss = 0.028
0174 (* 1 = 0.0280174 loss)
10327 19:37:19.633744 1028 solver.cpp:322] Optimization Done.
10327 19:37:19.634743 1028 caffe.cpp:223] Optimization Done.
C:\Users\Administrator\Desktop\caffe-master>
```

图6-5 Windows Caffe运行结束

生成的快照文件和保存的权值文件如图 6-6 所示。



名称	修改日期	类型	大小
lenet_consolidated_solver.prototxt	2016/3/27 17:46	PROTOTXT 文件	5 KB
lenet_iter_5000.caffemodel	2016/3/27 19:37	CAFFEMODEL 文...	1,685 KB
lenet_iter_5000.solverstate	2016/3/27 19:37	SOLVERSTATE ...	1,685 KB
lenet_iter_10000.caffemodel	2016/3/27 19:37	CAFFEMODEL 文...	1,685 KB
lenet_iter_10000.solverstate	2016/3/27 19:37	SOLVERSTATE ...	1,685 KB
lenet_multistep_solver.prototxt	2016/3/27 17:46	PROTOTXT 文件	1 KB

图6-6 快照文件和权值文件

可见，Caffe 在 Windows 环境下运行命令、过程和结果与 Linux 下相同。

6.3 回顾

通过今天内容我们可以初步了解到一个完整深度学习系统最核心的两个方面：**数据和模型**。数据是带标签的图片集，分训练集和测试集；模型是描述 CNN 结构的有向无环图（DAG），表示对原始数据的处理方式。

Caffe 并不是直接处理原始数据的，而是由预处理程序将原始数据变换存储为 LMDB 格式，这种方式可保持较高的 IO 效率，加快训练时的数据加载速度。模型通常用 ProtoBuffer 文本格式表述，训练结果保存为 ProtoBuffer 二进制文件或 HDF5 格式文件。深度学习的过程其实就是利用训练数据对模型进行训练，将数据中蕴藏的大量信息通过机器学习算法不断收集到模型中，利用训练好的模型对现实世界中的相似数据进行特定处理（如分类、识别、检测、定位）。

我们回顾一下今天使用的 build/tools/caffe.bin 用法。

```
$ ./build/tools/caffe.bin
caffe.bin: command line brew
usage: caffe <command><args>

commands:
  train          训练或微调一个模型
  test           对一个模型打分
  device_query   显示 GPU 诊断信息
  time           评估模型执行时间

Flags from tools/caffe.cpp:
```

-gpu (可选参数, 给定时运行在 GPU 模式, '-gpu all' 则表示运行在所有可用 GPU 设备上, 此时真正训练批量大小是 $N * B$, N 为指定的 GPU 设备数目)

-iterations (循环迭代次数, 默认为 50)

-model (指定模型定义文本文件名, *.prototxt)

-sighup_effect (当收到 SIGHUP 信号时要采取的动作, 可选项: snapshot、stop 或 none, 默认为 snapshot, 即打快照)

-sigint_effect (当收到 SIGINT 信号时要采取的动作, 可选项同上, 默认为 stop)

-snapshot (恢复训练时需要指定上次中止的快照, *.solverstate)

-solver (指定求解器文本文件名, *.prototxt)

-weights (指定用于微调的预训练权值, *.caffemodel, 不可与 snapshot 同时出现)

今天使用了该工具的 train 和 test 命令。后面在第 15 天内容中我们会用到另外两个命令。

我们学习了 Caffe 提供的简单例程, 目的是让初学者轻松上手, 其中最简单的两个例子是 examples/mnist/和 examples/cifar10/, 前者用于手写数字识别, 后者用于小图片分类。更复杂的例子位于 models/bvlc_reference_caffenet/、models/bvlc_googlenet/、models/bvlc_reference_rcnn_ilsvrc13/, 现在可能看似高深, 其实与简单模型在使用上并无二致, 只是更换了不同数据集、不同模型而已。

6.4 练习题

1. 写一个程序, 将你的手写体数字图片 (记得缩放到 28×28) 送入本节训练得到的 LeNet 模型中做预测, 评估数字识别效果。
2. 写一个程序, 将所有 LeNet-5 识别出错的样本导出, 观察并思考为什么机器会识别出错。
3. 按照本节步骤, 试着运行 Caffe CIFAR10 例程 (examples/cifar10), 分析输出日志。

6.5 参考资料

- [1] <http://yann.lecun.com/exdb/mnist/>
- [2] <http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf>
- [3] 百度云盘 (<http://pan.baidu.com/s/1sks4XFv>, 提取码: idi7)

篇尾语

上篇·初见的这几天我们主要熟悉了深度学习技术和开源框架 Caffe 的使用，掌握了 MNIST 数据集的细节和 LeNet-5 简单深度学习模型的使用方法。从这时开始你可能已经按捺不住想更多地了解 Caffe 的实现细节，希望与它有更多的“共同语言”。坠入爱河的感觉可能就是 这样吧~

中篇 热恋

昨夜星辰昨夜风，画楼西畔桂堂东。

身无彩凤双飞翼，心有灵犀一点通。

隔座送钩春酒暖，分曹射覆蜡灯红。

嗟余听鼓应官去，走马兰台类转蓬。

——李商隐《无题》

经过上篇初见 Caffe，也许你已经对其一见倾心，希望在本篇深入了解 Caffe 的每个模块如何实现。阅读源码是一项需要耐心和毅力的工作，如果不得要领，难以达到效果。Caffe 代码虽然更新很快，但主要框架变化很小，按照本篇的脉络结合适当的阅读工具可以轻松地学会在源码中遨游，与 Caffe “心有灵犀”。

第7天

Caffe 代码梳理

Caffe 是用 C++编写的深度学习框架，大量使用了类的封装、继承、多态，所以也可以用来学习 C++语言特性。Caffe 类数目众多，但通过面向对象编程（OOP）方式组织得很好，所以只要遵循类继承规则顺藤摸瓜，就不会看得云里雾里，迷失丛林。

7.1 Caffe 目录结构

在 Caffe 的根目录下执行 `tree` 命令查看 Caffe 目录结构：

```
$ tree -d
.
├── build -> .build_release      // 编译结果存放处，子目录结构与主目录类似
├── cmake                      // 使用 CMake 编译时会用到，不关注
│   ├── External
│   ├── Modules
│   └── Templates
├── data                      // 用于存放原始数据及数据获取脚本
│   ├── cifar10              // 存放 CIFAR10 小图片原始数据
│   ├── ilsvrc12             // 存放 ImageNet Meta 数据，原始数据需要另外下载
│   └── mnist                // 存放 MNIST 手写体数字图像数据
├── distribute                // 编译后生成发布包的位置，用于迁移
│   ├── bin
│   └── lib
├── docker                   // 同样是为了便于迁移，使用了 Docker 工具
│   └── standalone
│       ├── cpu
│       └── gpu
```

```

|   └── templates
├── docs                // doxygen 工程文件放在这里，可生成 Caffe ref_man.pdf
|   ├── _layouts
|   ├── images
|   ├── stylesheets
|   └── tutorial
|       └── fig
├── examples            // 存放 Caffe 简单例程
|   ├── cifar10         // CIFAR10 例程
|   ├── cpp_classification    // 图像分类例程
|   ├── feature_extraction  // 特征提取例程
|   ├── finetune_flickr_style // finetune 例程
|   ├── finetune_pascal_detection // finetune 例程
|   ├── hdf5_classification  // 使用 HDF5 数据源的分类例程
|   ├── imagenet         // ImageNet 例程，使用 bvlc_reference_caffenet 模型
|   ├── images
|   ├── mnist            // MNIST 手写体数字识别例程
|   |   ├── mnist_test_lmdb
|   |   └── mnist_train_lmdb
|   ├── net_surgery
|   ├── pycaffe
|   |   └── layers
|   ├── siamese
|   └── web_demo         // 一个 Web Server + 分类例程
|       └── templates
├── include              // Caffe 头文件集中存放于这个目录
|   ├── caffe
|   |   ├── layers
|   |   ├── test
|   |   └── util
├── matlab               // 适用于 Matlab 做 Wrapper，具体可以参考 RCNN 源码
|   ├── +caffe
|   |   ├── +test
|   |   ├── imagenet
|   |   └── private
|   ├── demo
|   └── hdf5creation
├── models               // 存放示例模型
|   └── bvlc_alexnet     // 经典的 AlexNet

```

```
|      ├── bvlc_googlenet           // GoogLeNet
|      ├── bvlc_reference_caffenet  // Caffe 模拟的 AlexNet
|      ├── bvlc_reference_rcnn_ilsvrc13 // RCNN 模型，参见参考资料[1]
|      └── finetune_flickr_style
└── python      // 用于 Python Wrapper
    ├── caffe
    │   ├── imagenet
    │   ├── proto
    │   └── test
└── scripts     // 存放脚本
    ├── travis
└── src         // Caffe 源码
    ├── caffe
    │   ├── layers      // 各个层的具体实现
    │   ├── proto       // proto 描述文件，学习数据结构先从这里开始
    │   ├── solvers
    │   ├── test
    │   │   └── test_data
    │   └── util
    └── gtest
└── tools       // 常用工具源码
└── extra
```

77 directories

我们需要关注三个子目录：include/、src/和 tools/。本篇将要分析的所有代码都出自这三个子目录。

7.2 如何有效阅读 Caffe 源码

如何有效阅读 Caffe 源码？这是多数初学者经常问的一个问题。

Caffe 源码阅读路线最好是从 `src/caffe/proto/caffe.proto` 开始, 了解基本数据结构内存对象和磁盘文件的一一映射关系 (如何从磁盘文件加载一个数据结构到内存对象, 以及如何将内存对象保存为磁盘文件, 这中间的过程其实都是由 `ProtoBuffer` 工具自动完成的)。

第二步是看头文件。不用急于去看 `cpp` 文件，先通过头文件类声明理解整个框架，发挥想象力去“猜”具体实现，从基类向派生类顺藤摸瓜看下去，很容易掌握这些类的使用方法。

第三步就是有针对性地去查看 `cpp` 和 `cu` 文件了。一般而言，Caffe 框架并不需要大改，按需求派生新的类即可。例如，你使用了新的卷积算法，需要自己实现相应的 `ConvolutionLayer`，则只需从已有的 `ConvolutionLayer` 派生一个新类 `MyConvolutionLayer`，然后将前向传播计算、反向传播计算按自己的算法实现即可。这一阶段关注点在算法实现上，相应的测试和正确性验证手段是必需的。

第四步就很自由了，可以编写各类工具，集成到 Caffe 内部。在 `tools/` 下面已有很多实用工具（如训练模型、测试模型、特征提取、转换数据格式等），可以根据需要修改。另外，也可以学习用 Python 或 Matlab 包装 Caffe 的方法，便于调节模型训练效果。

TIPS: Linux 使用技巧之阅读代码时如何快速追踪某个关键词?

方法一：打开多个终端，或用水平/垂直切分命令，将窗口切分成多份，将 `cpp` 文件包含的所有头文件都打开，利用 `vi` 的查找命令进行追踪。这种方法在工程较简单，头文件较少，或者对工程比较熟悉，了解每个头文件的内容时比较好用；而在工程较大，包含头文件较多，以及递归层次较多的场合下就不实用了。

方法二：使用 Linux `grep` 命令。例如，在 `src/caffe/layer_factory.cpp` 中有个宏调用：`REGISTER_LAYER_CREATOR(Pooling, GetPoolingLayer);`，看到这行时可能会比较困惑，这个宏到底干了什么呢？在 Caffe 根目录下运行以下命令：

```
$ grep -n -H -R "REGISTER_LAYER_CREATOR" *
include/caffe/layer_factory.hpp:34:*REGISTER_LAYER_CREATOR(MyAwesome, GetMyAwesomeLayer)
include/caffe/layer_factory.hpp:113:#define REGISTER_LAYER_CREATOR(type, creator) \
include/caffe/layer_factory.hpp:123:REGISTER_LAYER_CREATOR(type, Creator_##type##Layer)
src/caffe/layer_factory.cpp:41:REGISTER_LAYER_CREATOR(Convolution,
GetConvolutionLayer);
src/caffe/layer_factory.cpp:71:REGISTER_LAYER_CREATOR(Pooling, GetPoolingLayer);
src/caffe/layer_factory.cpp:94:REGISTER_LAYER_CREATOR(ReLU, GetReLULayer);
src/caffe/layer_factory.cpp:117:REGISTER_LAYER_CREATOR(Sigmoid, GetSigmoidLayer);
src/caffe/layer_factory.cpp:140:REGISTER_LAYER_CREATOR(Softmax, GetSoftmaxLayer);
src/caffe/layer_factory.cpp:163:REGISTER_LAYER_CREATOR(TanH, GetTanHLayer);
src/caffe/layer_factory.cpp:179:REGISTER_LAYER_CREATOR(Python, GetPythonLayer);
```

命令行参数解释：

- `-n` —显示行号，便于定位
- `-H` —显示文件名，便于定位
- `-R` —递归查找每个子目录，适合工程较大、分多个目录存放的场景

使用方法二的好处很多，首先非常直观地显示了所有包含这个宏的文件名和行号，我们能够通过输出仔细甄别出宏定义位置。另外，无须分别打开每个文件，也能看到整个工程中所有通过该宏注册的层生成器（卷积层，下采样层，非线性层 Sigmoid、ReLU 和 TanH，分类层 Softmax，以及 Python 层）。打开 include/caffe/layer_factory.hpp，跳到第 113 行可以看到该宏的定义：

```
#define REGISTER_LAYER_CREATOR(type, creator) \
    static LayerRegisterer<float> g_creator_f_##type(#type, creator<float>);\
    static LayerRegisterer<double> g_creator_d_##type(#type, creator<double>)
```

利用这种方法，可以很容易地在 Caffe 源码中定位下一节将要讲述的内容。

7.3 Caffe 支持哪些深度学习特性

深度学习一直处于不断发展的状态，Caffe 也在不断地适应深度学习的发展。我们先看一个简化的深度学习前馈卷积网络（CNN）模型，如图 7-1 所示。

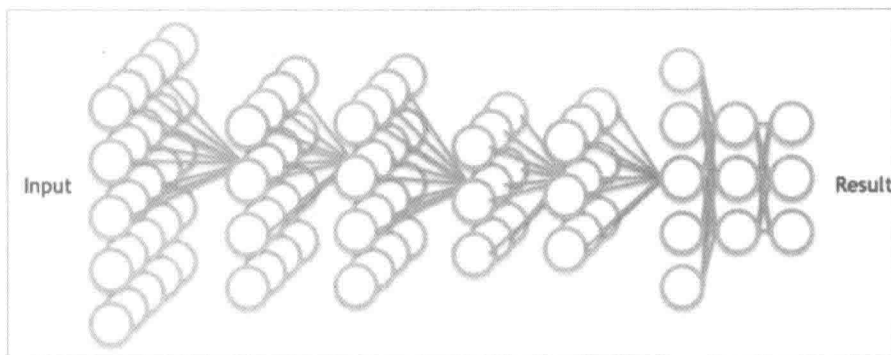


图7-1 CNN模型

在上面的 CNN 模型中，输入层为二维图像数据，前面几层都是卷积层（Convolutional Layer），用于提取低维到高维特征；最后两层为全连接层，类似于多层感知器（MLP），用于对前面提供的特征进行分类。卷积层和全连接层我们统称为权值层，因为这两种层都具有可学习参数（权值），是网络训练的对象。我们首先介绍这两种层的具体实现。

7.3.1 卷积层

卷积是大自然中最常见的运算，一切信号观测、采集、传输、处理都可以用卷积过程实现。例如，你拍照时手抖了一下，导致照片模糊，实际上等价于手没抖拍摄的正常照片与一个表示

手抖的卷积核进行卷积运算得到的结果。用公式表达如下：

$$\begin{aligned}
 Y(m,n) &= X(m,n) * H(m,n) \\
 &= \sum_{i=-\infty}^{+\infty} \sum_{j=-\infty}^{+\infty} X(i,j) H(m-i, n-j) \\
 &= \sum_{i=-\infty}^{+\infty} \sum_{j=-\infty}^{+\infty} X(m-i, n-j) H(i,j)
 \end{aligned}$$

其中， $X(m,n)$ 表示正常照片， $H(m,n)$ 表示手抖卷积核， $Y(m,n)$ 表示模糊的照片。如果能通过某种方法从 $Y(m,n)$ 估计出 $H(m,n)$ ，就可以采取措施恢复 $X(m,n)$ 实现相机防抖功能。更多卷积理论研究见参考资料[2]。

卷积层是卷积神经网络（CNN）典型层，CNN名称就是由该层冠名的，可见其重要性。卷积层计算步骤与上面公式定义的二维卷积稍有差异，首先是维度升至三维、四维卷积，与二维卷积相比增加了多个“通道”（channel），每个通道仍然按照二维卷积方式计算，多个通道与多个卷积核分别进行二维卷积，得到多通道输出，需要“合并”为一个通道；其次是卷积核在卷积计算时没有“翻转”，而是与输入图片做滑动窗口“相关”计算。卷积和相关的区别，请读者翻阅参考资料[3]深入研究。用公式重新表达为：

$$\begin{aligned}
 Y^l(m,n) &= X^k(m,n) \star H^{kl}(m,n) \\
 &= \sum_{k=0}^{K-1} \sum_{i=0}^{I-1} \sum_{j=0}^{J-1} X^k(m+i, n+j) H^{kl}(i,j)
 \end{aligned}$$

这里假定卷积层有 L 个输出通道和 K 个输入通道，于是需要有 LK 个卷积核实现通道数目的转换。其中 X^k 表示第 $k(0 \leq k < K)$ 个输入通道的二维特征图， Y^l 表示第 $l(0 \leq l < L)$ 个输出通道的二维特征图， H^{kl} 表示第 l 列、第 k 行二维卷积核。假定卷积核大小为 $I \cdot J$ ，每个输出通道的特征图大小均为 $M \cdot N$ ，则该层每个样本做一次前向传播时卷积层计算量为：

$$\text{Calculations(MAC)} = I \cdot J \cdot M \cdot N \cdot K \cdot L$$

以 LeNet-5 第二个卷积层为例，我们找到与上面公式对应的参数。首先从 examples/mnist/lenet_train_val.prototxt 中找到卷积参数描述：

```
convolution_param {
  num_output: 50 // 对应公式中的 L
  kernel_size: 5 // 对应公式中的 I 和 J
  stride: 1
  weight_filler {
```

```

    type: "xavier"
  }
  bias_filler {
    type: "constant"
  }
}

```

剩下的三个参数 K, M, N 从哪里找呢？答案是日志文件。

```

I0404 15:16:14.404001 1965830144 net.cpp:148] Top shape: 64 20 12 12 (184320)
I0404 15:16:14.404019 1965830144 net.cpp:156] Memory required for data: 3887360
I0404 15:16:14.404036 1965830144 layer_factory.hpp:77] Creating layer conv2
I0404 15:16:14.404063 1965830144 net.cpp:91] Creating Layer conv2
I0404 15:16:14.404078 1965830144 net.cpp:425] conv2 <- pool1
I0404 15:16:14.404098 1965830144 net.cpp:399] conv2 -> conv2
I0404 15:16:14.404726 1965830144 net.cpp:141] Setting up conv2
I0404 15:16:14.404755 1965830144 net.cpp:148] Top shape: 64 50 8 8 (204800)

```

通过日志文件中 conv2 这一层构建时的上下文信息可以获得特征图尺寸和前一层特征图通道数信息。如上面日志中粗体字部分显示，对应的值为：

$$M=N=8$$

$$K=20$$

由此可以计算出 LeNet-5 conv2 的单样本前向传播计算量为：

$$\begin{aligned}
 \text{Calculations(MAC)} &= 5 \cdot 5 \cdot 8 \cdot 8 \cdot 50 \cdot 20 \\
 &= 1\,600\,000 \text{MAC}
 \end{aligned}$$

注意实际中通常一次送入一批样本 (batch)，此时计算量应再乘上批量尺寸。

通过上述参数，还可以计算出卷积层的学习参数数量，其实非常简单，就是卷积核数目乘卷积核尺寸：

$$\begin{aligned}
 \text{Params} &= I \cdot J \cdot K \cdot L \\
 &= 5 \cdot 5 \cdot 50 \cdot 20 \\
 &= 25\,000
 \end{aligned}$$

这里定义计算量-参数量之比为 CPR (Calculations to Parameters Ratio)：

$$\text{CPR} = \text{Calculations} / \text{Params} = M \cdot N = 64$$

可以得出结论：卷积层的输出特征图尺寸越大，CPR 值越大，参数重复利用率越高。若输

入一批数据 (B 个样本), 则 CPR 值可再提高 B 倍。

Caffe 中卷积层的具体实现, 可以用上节介绍的方法查找 ConvolutionLayer 关键词获得相关的文件列表, 再按照先头文件、后 cpp 源文件、最后 cu 文件的顺序依次阅读。

7.3.2 全连接层

其实在卷积网络出现之前, 最早的深度学习网络计算类型都是全连接形式的。如图 7-2 所示的 DNN 模型, 也可以被解读为 “Dense Neural Networks”。

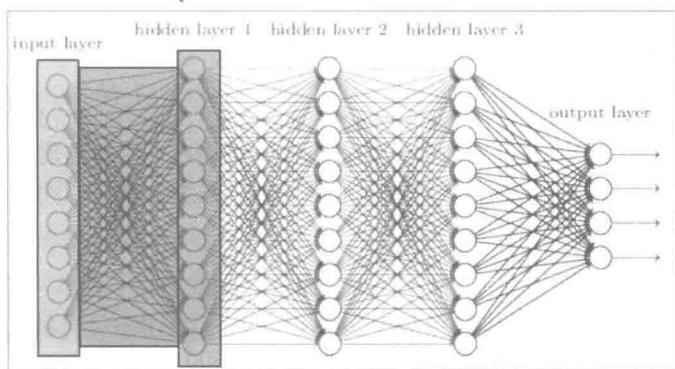


图7-2 DNN模型

从图 7-2 可以看到, 每个节点与相邻层的所有节点都有连接关系, 这是 “全连接层” 名称的由来。

全连接层的主要计算类型为矩阵-向量乘 (GEMV)。假设输入节点组成的向量为 \mathbf{x} , 维度为 D , 输出节点组成的向量为 \mathbf{y} , 维度为 V , 则全连接层计算可以表示为:

$$\mathbf{y} = W\mathbf{x}$$

其中, W 为 $V \cdot D$ 维权值矩阵。

我们分析一下 LeNet-5 第一个全连接层的计算量和参数量情况。在 examples/mnist/lenet_train_val.prototxt 中找到全连接参数描述:

```
inner_product_param {    // 全连接层参数
    num_output: 500      // 该层输出元素个数为 500, 对应公式中的 V
    weight_filler {
        type: "xavier"
    }
    bias_filler {
```

```

    type: "constant"
  }
}

```

运行日志：

```

I0404 15:16:14.404871 1965830144 net.cpp:148] Top shape: 64 50 4 4 (51200)
I0404 15:16:14.404881 1965830144 net.cpp:156] Memory required for data: 4911360
I0404 15:16:14.404888 1965830144 layer_factory.hpp:77] Creating layer ip1
I0404 15:16:14.404901 1965830144 net.cpp:91] Creating Layer ip1
I0404 15:16:14.404911 1965830144 net.cpp:425] ip1 <- pool2
I0404 15:16:14.404925 1965830144 net.cpp:399] ip1 -> ip1
I0404 15:16:14.409618 1965830144 net.cpp:141] Setting up ip1
I0404 15:16:14.409646 1965830144 net.cpp:148] Top shape: 64 500 (32000)

```

可以看到前一层的维度信息，而在全连接层中，直接将前一层输出展开为一维向量（batch 维度除外），所以 ip1 层对应的输入节点数目为 $D = 50 \cdot 4 \cdot 4 = 800$ 。

全连接层单样本前向传播计算量统计为：

$$\begin{aligned}
 \text{CalculationsMAC} &= V \cdot D \\
 &= 800 \cdot 500 \\
 &= 400\,000
 \end{aligned}$$

参数量统计为：

$$\begin{aligned}
 \text{Params} &= V \cdot D \\
 &= 800 \cdot 500 \\
 &= 400\,000
 \end{aligned}$$

CPR 值为：

$$\text{CPR} = \text{Calculations} / \text{Params} = 1$$

可见，全连接层的 CPR 值始终为 1，与输入、输出维度无关。从这个角度来说，单样本前向传播计算时，权值重复利用率很低。

将一批（ B 个）样本 \mathbf{x}_i 逐列拼接成矩阵 \mathbf{X} ，一次性通过全连接层，得到一批输出向量构成的矩阵 \mathbf{Y} ，称作批处理。相应地，前面的矩阵-向量乘运算升为矩阵-矩阵乘计算（GEMM）：

$$\mathbf{Y} = \mathbf{W}\mathbf{X}$$

这样全连接层前向计算量提高了 B 倍，而参数量不变，因此 CPR 提高了 B 倍。权值矩阵在多

样本之间实现了重用（共享），可提高计算速度。Caffe 中全连接层实现可以通过 InnerProductLayer 关键词查阅。

与 7.3.1 节的卷积层相比，全连接层参数量是其 16 倍，而计算量只有其 25%。如果输出特征图尺寸相同（ $M \cdot N = V$ ），卷积层的 CPR 值为全连接层的 $M \cdot N$ 倍。也就是说，卷积层在输出特征图维度实现了权值共享。这是降低参数量的重要举措。与此同时，卷积层局部连接特性（相比全连接）也大幅减少了参数量。这使得 CNN 网络中前几层卷积层参数量占比小，计算量占比大；而后几层全连接层参数量占比大，计算量占比小。大多数 CNN 模型都符合这个特点^[4]。因此我们在进行计算加速优化时，重点放在卷积层；而在进行参数优化、权值剪裁时，重点放在全连接层。

7.3.3 激活函数

神经网络之所以具有丰富的表达能力，除了“深”之外，还有一个重要因素即非线性处理单元，称为激活函数（Activation Function）或挤压函数（Squashing Function）。本节将关注如何在 Caffe 中的非线性层实现这些激活函数。

如图 7-3 所示的神经元模型， $\varphi(\cdot)$ 即为激活函数，实现将来自前一层的输入线性组合结果 u_k 动态范围压缩到特定值域（如 $[-1, 1]$ ）。具备非线性处理单元的神经网络（ ≥ 3 层），理论上可以逼近任意函数^{[5][6]}。

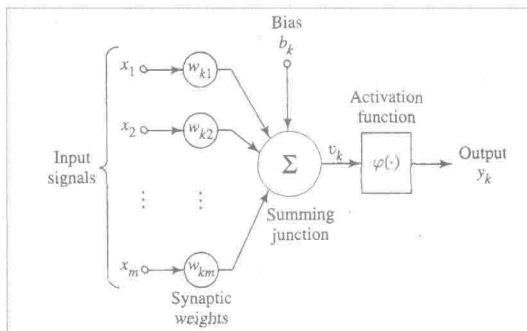


图7-3 神经元模型

几个常用的激活函数如下：

(1) Sigmoid 函数，值域为 $(0, 1)$ ，见图 7-4。

$$\varphi(x) = \frac{1}{1 + e^{-ax}}$$

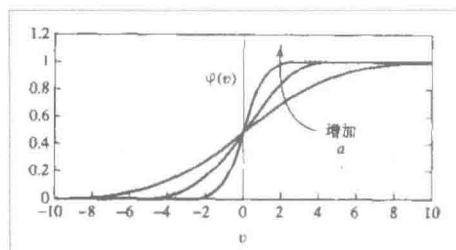


图7-4 Sigmoid函数

(2) tanh 函数，值域为 $(-1, 1)$ ，见图 7-5 所示。

$$\varphi(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

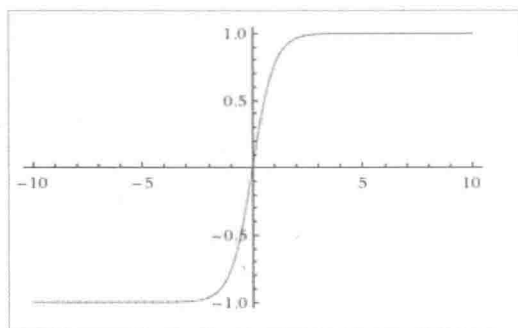


图7-5 tanh函数

(3) ReLU (Rectified Linear Unit, 规整化线性单元) 函数^[7]，值域为 $[0, +\infty)$ ，是一种非饱和和激活函数，见图 7-6。

$$\varphi(x) = \max(0, x)$$

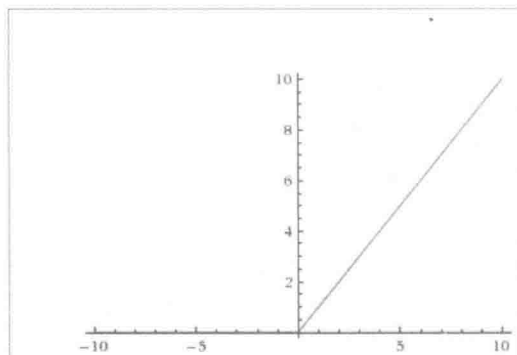


图7-6 ReLU函数

神经网络中最大的问题是梯度消失问题 (Gradient Vanishing Problem)，这在使用 Sigmoid、tanh 等饱和激活函数情况下尤为严重 (神经网络进行误差反向传播时，各层都要乘以激活函数的一阶导数 $G = e \cdot \phi'(x) \cdot x$ ，梯度每传递一层都会衰减一次，网络层数较多时，梯度 G 就会不停地衰减直至消失)，使得训练网络时收敛极慢，而 ReLU 这类非饱和激活函数收敛速度则快很多^[8]。本书使用的深度学习网络模型中大部分激活函数都选择 ReLU。

了解了三个典型的非线性激活函数后，接下来我们学习在 Caffe 中如何用代码实现对应层的计算，包括前向传播计算和反向传播计算。Caffe 的所有与激活函数相关的 Layer 类声明都位于 include/caffe/neural_layers.hpp 中，本书将它们统称为非线性层，我们重点关注 ReLULayer、SigmoidLayer 和 TanHLayer 这三类。

在前面的 MNIST 例程中 LeNet-5 模型使用了 ReLU 层，在 examples/mnist/lenet_train_val.prototxt 中找到该层的描述：

```
layer { // 新的非线性层，用 ReLU 方法
  name: "relu1"
  type: "ReLU"
  bottom: "i1"
  top: "i1"
}
```

与卷积层、全连接层最大的不同，就是没有权值相关的参数，描述相对简单。另外两种层没有实际样例，怎么办呢？这时按照我们的 Caffe 源码阅读方法论，从 src/caffe/proto/caffe.proto 中获得灵感。

```
// ReLU 层参数
message ReLUParameter {
  // Leaky ReLU 参数，我们暂不关心
  optional float negative_slope = 1 [default = 0];
  enum Engine { // 计算引擎选择
    DEFAULT = 0;
    CAFFE = 1; // Caffe 实现
    CUDNN = 2; // CUDNN 实现
  }
  optional Engine engine = 2 [default = DEFAULT];
}

// Sigmoid 层参数
message SigmoidParameter {
  enum Engine { // 计算引擎选择
```

```

    DEFAULT = 0;
    CAFFE = 1;    // Caffe 实现
    CUDNN = 2;    // CUDNN 实现
}
optional Engine engine = 1 [default = DEFAULT];
}
// tanh 层参数
message TanHParameter {
    enum Engine {    // 计算引擎选择
        DEFAULT = 0;
        CAFFE = 1;    // Caffe 实现
        CUDNN = 2;    // CUDNN 实现
    }
    optional Engine engine = 1 [default = DEFAULT];
}

```

非线性层的共同特点就是对前一层 blob 中的数值逐一进行非线性变换，并放回原 blob 中。在 include/caffe/neural_layers.hpp 中查看类声明如下：

```

// 非线性层的鼻祖 NeuronLayer，派生于 Layer 类，特点是输出 blob (y) 与输入 blob (x) 尺寸相同
template <typename Dtype>
class NeuronLayer : public Layer<Dtype> {
public:
    explicit NeuronLayer(const LayerParameter& param)
        : Layer<Dtype>(param) {}
    virtual void Reshape(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);

    virtual inline int ExactNumBottomBlobs() const { return 1; }
    virtual inline int ExactNumTopBlobs() const { return 1; }
};

// ReLULayer，派生于 NeuronLayer，实现了 ReLU 激活函数计算
template <typename Dtype>
class ReLULayer : public NeuronLayer<Dtype> {
public:
    // 显式构造函数
    explicit ReLULayer(const LayerParameter& param)
        : NeuronLayer<Dtype>(param) {}
    // 返回类名字字符串
    virtual inline const char* type() const { return "ReLU"; }
}

```

```
protected:
    // 前向传播函数
    virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    // 反向传播函数
    virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);
    virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);
};
```

// SigmoidLayer, 派生于 NeuronLayer, 实现了 Sigmoid 激活函数计算

```
template <typename Dtype>
class SigmoidLayer : public NeuronLayer<Dtype> {
public:
    // 显式构造函数
    explicit SigmoidLayer(const LayerParameter& param)
        : NeuronLayer<Dtype>(param) {}
    // 返回类名字符串
    virtual inline const char* type() const { return "Sigmoid"; }
```

```
protected:
    // 前向传播函数
    virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    // 反向传播函数
    virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);
    virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);
};
```

// TanHLayer, 派生于 NeuronLayer, 实现了 tanh 激活函数计算

```
template <typename Dtype>
class TanHLayer : public NeuronLayer<Dtype> {
public:
```

```

// 显式构造函数
explicit TanHLayer(const LayerParameter& param)
    : NeuronLayer<Dtype>(param) {}
// 返回类名字符串
virtual inline const char* type() const { return "TanH"; }

protected:
// 前向传播函数
virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top);
virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top);
// 反向传播函数
virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);
virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);
};

```

比较简单，各自声明了相应的 Forward 和 Backward 函数。下面深入到这些函数的实现中。首先看 src/caffe/layers/relu_layer.cpp 中前向传播函数的实现代码。

```

template <typename Dtype>
void ReLULayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
const vector<Blob<Dtype>*>& top) {
    // (只读) 获得输入 blob 的 data 指针
    const Dtype* bottom_data = bottom[0]->cpu_data();
    // (读写) 获得输出 blob 的 data 指针
    Dtype* top_data = top[0]->mutable_cpu_data();
    // 获得输入 blob 元素个数
    const int count = bottom[0]->count();
    // Leaky ReLU 参数，从 layer_param_ 中获得，默认为 0，即普通 ReLU
    Dtype negative_slope = this->layer_param_.relu_param().negative_slope();
    // 执行 ReLU 操作。我们姑且认为 negative_slope 值为 0，不考虑 Leaky ReLU
    for (int i = 0; i < count; ++i) {
        top_data[i] = std::max(bottom_data[i], Dtype(0))
            + negative_slope * std::min(bottom_data[i], Dtype(0));
    }
}

```


不出所料，用一层 for 循环就搞定了。下面看反向传播函数的实现代码。

```
template <typename Dtype>
void ReLULayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
    // 如果需要做反向传播计算
    if (propagate_down[0]) {
        // (只读) 获得前一层的 data 指针
        const Dtype* bottom_data = bottom[0]->cpu_data();
        // (只读) 获得后一层的 diff 指针
        const Dtype* top_diff = top[0]->cpu_diff();
        // (读写) 获得前一层的 diff 指针
        Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
        // 获得需要参与计算的元素总数
        const int count = bottom[0]->count();
        // Leaky ReLU 参数，姑且认为是 0
        Dtype negative_slope = this->layer_param_.relu_param().negative_slope();
        for (int i = 0; i < count; ++i) {
            // ReLU 的导函数就是 (bottom_data[i] > 0)，根据求导链式法则，后一层的误差乘以导函数得到前一层
            // 的误差
            bottom_diff[i] = top_diff[i] * ((bottom_data[i] > 0)
                + negative_slope * (bottom_data[i] <= 0));
        }
    }
}
```

从上面代码可以看到 ReLU 计算非常简单。下面看 SigmoidLayer 的实现，位于 src/caffe/layers/sigmoid_layer.cpp 中：

```
// 实现 Sigmoid 函数，这里将 7.3.3 节中 Sigmoid 函数中的参数 a 固定设置为 1
template <typename Dtype>
inline Dtype sigmoid(Dtype x) {
    return 1. / (1. + exp(-x));
}

// 前向传播函数
template <typename Dtype>
void SigmoidLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
    const Dtype* bottom_data = bottom[0]->cpu_data();
```

```

Dtype* top_data = top[0]->mutable_cpu_data();
const int count = bottom[0]->count();
for (int i = 0; i < count; ++i) {
    top_data[i] = sigmoid(bottom_data[i]);
}
}
// 反向传播函数
template <typename Dtype>
void SigmoidLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
    if (propagate_down[0]) {
        const Dtype* top_data = top[0]->cpu_data();
        const Dtype* top_diff = top[0]->cpu_diff();
        Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
        const int count = bottom[0]->count();
        for (int i = 0; i < count; ++i) {
            // top_data 中是前向传播阶段计算结果  $\phi(x)$ ，这里重用，可以降低计算量
            const Dtype sigmoid_x = top_data[i];
            // Sigmoid 函数的导函数是  $\phi'(x) = \phi(x) \cdot (1 - \phi(x))$ ，根据求导链式法则，后一层的误差乘上导函数即得到前
            // 一层的误差
            bottom_diff[i] = top_diff[i] * sigmoid_x * (1. - sigmoid_x);
        }
    }
}

```

TanHLayer 的实现位于 src/caffe/layers/tanh_layer.cpp 中：

```

// 前向传播函数
template <typename Dtype>
void TanHLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
    const Dtype* bottom_data = bottom[0]->cpu_data();
    Dtype* top_data = top[0]->mutable_cpu_data();
    const int count = bottom[0]->count();
    for (int i = 0; i < count; ++i) {
        top_data[i] = tanh(bottom_data[i]);
    }
}
// 反向传播函数

```

```

template <typename Dtype>
void TanHLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
    if (propagate_down[0]) {
        const Dtype* top_data = top[0]->cpu_data();
        const Dtype* top_diff = top[0]->cpu_diff();
        Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
        const int count = bottom[0]->count();
        Dtype tanhx;
        for (int i = 0; i < count; ++i) {
            // top_data 中是前向传播阶段计算结果  $\phi(x)$ ，这里重用，可以降低计算量
            tanhx = top_data[i];
            // tanh 函数的导函数是  $\phi'(x) = (1 - \phi^2(x))$ ，根据求导链式法则，后一层的误差乘上导函数即得到前一层的误差
            bottom_diff[i] = top_diff[i] * (1 - tanhx * tanhx);
        }
    }
}

```

可见，非线性层虽然公式表示较为复杂，但代码实现都非常简洁、直观，只要掌握了基本求导技巧，读者同样可以推导出非线性层其他类的反向传播公式。读者可试着做做练习题 3。

7.4 小结

今天我们从整体上对 Caffe 做了一个概要预览，掌握了使用关键词搜索阅读源码的方法。通过了解一些组织良好、开源的框架实现方式，学习其优点，并熟练应用到自己的实际开发中，将会让你成长为优秀的软件架构师。

7.5 练习题

1. 在 Caffe 源码中查找以下宏的原始定义：

```

NOT_IMPLEMENTED
INstantiate_Class

```

2. C++ 什么时候使用虚函数？什么时候使用类模板？什么时候使用虚基类？

3. 推导 BNLLayer 的误差反向传播公式，其激活函数为：

$$\varphi(x) = \begin{cases} x + \log(1 + e^{-x}), & x > 0 \\ \log(1 + e^x), & \text{否则} \end{cases}$$

答案：自己阅读 `src/caffe/layers/bnll_layer.cpp`。

提示：Caffe 源码提供的并不是最优算法，可以自己修改。

7.6 参考资料

[1] <https://github.com/rbgirshick/rcnn>

[2] 邹谋炎，反卷积和信号复原，国防工业出版社，2001

[3] 郑君里等，信号与系统（上下册），高等教育出版社，2011.3

[4] Alex Krizhevsky, One weird trick for parallelizing convolutional neural networks, arXiv:1404.5997v2

[5] G. Cybenko. Approximation by superpositions of a sigmoidal function

[6] Kurt Hornik. Multilayer feedforward networks are universal approximators

[7] Maas, A. L. Rectifier nonlinearities improve neural network acoustic models

[8] Alex Krizhevsky, ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012

第 8 天

Caffe 数据结构

今天我们来一起阅读 Caffe 最基础的数据结构源码。在 Caffe 中一个 CNN 模型使用 Net 表示，而 Net 是由多个 Layer 堆叠而成的，如图 8-1 所示。

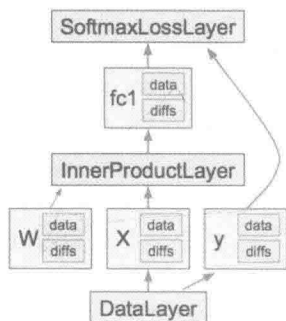


图8-1 Caffe数据结构

荀子《劝学》：不积跬步，无以至千里；不积小流，无以成江海。Caffe 的万丈高楼（Net）是按照我们设计的图纸（prototxt），用 Blob 这些砖块筑成一层层（Layer）楼房，最后通过 SGD 方法（Solver）进行简装修（Train）、精装修（Finetune）实现的。今天将向读者介绍 Caffe 大厦的砖石结构——Blob、Layer、Net 的基本概念。

8.1 Blob

Caffe 使用称为 Blob 的 4 维数组用于存储和交换数据。Blob 提供了统一的存储器接口，持有一批图像或其他数据、权值、权值更新值。其他深度学习框架也有与 Blob 对应的数据结构，如 Torch/Theano/TensorFlow 中的 Tensor、MxNet 中的 NDArray、cuda-convnet 中的 NVMatrix 等。

Blob 在内存中表示 4 维数组，维度从低到高为 (width_, height_, channels_, num_)，如果不通就当作视频流吧，width_ 和 height_ 表示图像的宽和高，channels_ 表示颜色通道 RGB，num_ 表示第几帧，用于存储数据或权值 (data) 和权值增量 (diff)，在进行网络计算时，每层的输入、输出都需要通过 Blob 对象缓冲。Blob 是 Caffe 的基本存储单元，下面就从该类入手开始学习。

8.1.1 Blob 基本用法

为了更好地理解 Caffe 源码，我们先建立对 Blob 的感性认识，学会如何使用 Blob，“猜测”其实现过程，最后再通过阅读源码获得答案。

在使用 Blob 之前，需要先包含头文件“#include <caffe/blob.hpp>”，再通过“using namespace caffe;”使用命名空间 caffe。不熟悉 C++ 的同学，建议手边放一本《21 天学通 C++》^[1]与本书一起看，效果更佳 (^_^)。

Blob 是一个模板类，所以创建对象时需要制定模板参数。写一个简单测试程序 blob_demo.cpp:

```
#include <vector>
#include <iostream>
#include <caffe/blob.hpp>
using namespace caffe;
using namespace std;
int main(void)
{
    Blob<float> a;
    cout<<"Size : "<< a.shape_string()<<endl;
    a.Reshape(1, 2, 3, 4);
    cout<<"Size : "<< a.shape_string()<<endl;
    return 0;
}
```

如上代码首先创建了整型 Blob 对象 a，打印其维度信息，然后调用其 Reshape() 方法，再次打印其维度信息。

使用如下命令编译这个程序（假设之前已经编译好的 Caffe 位于 \$CAFFE_ROOT）:

```
$ g++ -o app blob_demo.cpp -I $CAFFE_ROOT/include/ -D CPU_ONLY \
-I $CAFFE_ROOT/.build_release/src/ -L $CAFFE_ROOT/build/lib/ -lcaffe
```

生成了可执行程序 app。

运行该程序：

```
$ export LD_LIBRARY_PATH= $CAFFE_ROOT/build/lib/:$LD_LIBRARY_PATH
$ ./app
Size : (0)
Size : 1 2 3 4 (24)
```

创建 Blob 对象之后，可以通过 mutable_cpu[gpu]_data[diff]函数修改其内部数值：

```
// 续上面代码
float * p = a.mutable_cpu_data();
for(int i = 0; i < a.count(); i++)
{
    p[i] = i;
}
for(int u = 0; u < a.num(); u++)
{
    for(int v = 0; v < a.channels(); v++)
    {
        for(int w = 0; w < a.height(); w++)
        {
            for(int x = 0; x < a.width(); x++)
            {
                cout<<"a["<<u<<"["<<v<<"["<<w<<"["<<x<<" = "<< a.data_at(u, v, w, x)<<endl;
            }
        }
    }
}
```

运行结果如下：

```
a[0][0][0][0] = 0
a[0][0][0][1] = 1
a[0][0][0][2] = 2
a[0][0][0][3] = 3
a[0][0][1][0] = 4
a[0][0][1][1] = 5
a[0][0][1][2] = 6
a[0][0][1][3] = 7
```

```

a[0][0][2][0] = 8
a[0][0][2][1] = 9
a[0][0][2][2] = 10
a[0][0][2][3] = 11
a[0][1][0][0] = 12
a[0][1][0][1] = 13
a[0][1][0][2] = 14
a[0][1][0][3] = 15
a[0][1][1][0] = 16
a[0][1][1][1] = 17
a[0][1][1][2] = 18
a[0][1][1][3] = 19
a[0][1][2][0] = 20
a[0][1][2][1] = 21
a[0][1][2][2] = 22
a[0][1][2][3] = 23

```

可见，Blob 下标访问与 C/C++ 高维数组几乎一致，而 Blob 的强大之处在于可以自动同步 CPU/GPU 上的数据。

Blob 还支持计算所有元素绝对值之和（L1-范数）、平方和（L2-范数）：

// 续上面代码

```

cout<<"ASUM = "<<a.asum_data()<<endl;
cout<<"SUMSQ = "<<a.sumsq_data()<<endl;

```

输出结果为：

```

ASUM = 276
SUMSQ = 4324

```

除了 data，我们还可以改 diff 部分，与 data 操作基本一致：

```

#include <vector>
#include <iostream>
#include <caffe/blob.hpp>
using namespace caffe;
using namespace std;
int main(void)
{
    Blob<float> a;
    cout<<"Size : "<< a.shape_string()<<endl;

```



```

a.Reshape(1, 2, 3, 4);
cout<<"Size : "<< a.shape_string()<<endl;
float * p = a.mutable_cpu_data();
float * q = a.mutable_cpu_diff();
for(int i = 0; i < a.count(); i++)
{
    p[i] = i; // 将 data 初始化为 1, 2, 3 ...
    q[i] = a.count() - 1 - i; // 将 diff 初始化为 23, 22, 21, ...
}
a.Update(); // 执行 Update 操作, 将 diff 与 data 融合
            // 这也是 CNN 权值更新步骤的最终实施者
for(int u = 0; u < a.num(); u++)
{
    for(int v = 0; v < a.channels(); v++)
    {
        for(int w = 0; w < a.height(); w++)
        {
            for(int x = 0; x < a.width(); x++)
            {
                cout<<"a["<<u<<"]["<<v<<"]["<<w<<"]["<<x<<"] = "<< a.data_at(u, v, w, x)<<endl;
            }
        }
    }
}

cout<<"ASUM = "<<a.asum_data()<<endl;
cout<<"SUMSQ = "<<a.sumsq_data()<<endl;

return 0;
}

```

编译后运行输出如下:

```

$ ./app
Size : {0}
Size : 1 2 3 4 (24)
a[0][0][0][0] = -23
a[0][0][0][1] = -21
a[0][0][0][2] = -19
a[0][0][0][3] = -17
a[0][0][1][0] = -15

```

```

a[0][0][1][1] = -13
a[0][0][1][2] = -11
a[0][0][1][3] = -9
a[0][0][2][0] = -7
a[0][0][2][1] = -5
a[0][0][2][2] = -3
a[0][0][2][3] = -1
a[0][1][0][0] = 1
a[0][1][0][1] = 3
a[0][1][0][2] = 5
a[0][1][0][3] = 7
a[0][1][1][0] = 9
a[0][1][1][1] = 11
a[0][1][1][2] = 13
a[0][1][1][3] = 15
a[0][1][2][0] = 17
a[0][1][2][1] = 19
a[0][1][2][2] = 21
a[0][1][2][3] = 23
ASUM = 288
SUMSQ = 4600

```

可见，在 Update() 函数中，实现了 $\text{data} = \text{data} - \text{diff}$ 操作。这个在 CNN 权值更新时会用到，我们后面会深入研究。

将 Blob 内部值保存磁盘，或者从磁盘载入内存，可以分别通过 ToProto()、FromProto() 实现：

```

// 续上面代码
// .....
#include <caffe/util/io.hpp> // 需要包含这个头文件
// .....

BlobProto bp; // 构造一个 BlobProto 对象
a.ToProto(&bp, true); // 将 a 序列化，连同 diff（默认不带）
WriteProtoToBinaryFile(bp, "a.blob"); // 写入磁盘文件 "a.blob"
BlobProto bp2; // 构造一个新的 BlobProto 对象
ReadProtoFromBinaryFileOrDie("a.blob", &bp2); // 读取磁盘文件
Blob<float> b; // 新建一个 Blob 对象 b
b.FromProto(bp2, true); // 从序列化对象 bp2 中克隆 b（连同形状）

```

```

for(int u = 0; u < b.num(); u++)
{
    for(int v = 0; v < b.channels(); v++)
    {
        for(int w = 0; w < b.height(); w++)
        {
            for(int x = 0; x < b.width(); x++)
            {
                // 打印 b
                cout<<"b["<<u<<"["<<v<<"["<<w<<"["<<x<<" = "<< b.data_at(u, v, w, x)<<endl;
            }
        }
    }
}

```

编译时加入 “-lglog -lboost_system” 选项。运行输出如下：

```

$ ./app
Size : (0)
Size : 1 2 3 4 (24)
a[0][0][0][0] = -23
a[0][0][0][1] = -21
a[0][0][0][2] = -19
a[0][0][0][3] = -17
a[0][0][1][0] = -15
a[0][0][1][1] = -13
a[0][0][1][2] = -11
a[0][0][1][3] = -9
a[0][0][2][0] = -7
a[0][0][2][1] = -5
a[0][0][2][2] = -3
a[0][0][2][3] = -1
a[0][1][0][0] = 1
a[0][1][0][1] = 3
a[0][1][0][2] = 5
a[0][1][0][3] = 7
a[0][1][1][0] = 9
a[0][1][1][1] = 11
a[0][1][1][2] = 13
a[0][1][1][3] = 15

```

```

a[0][1][2][0] = 17
a[0][1][2][1] = 19
a[0][1][2][2] = 21
a[0][1][2][3] = 23
ASUM = 288
SUMSQ = 4600
b[0][0][0][0] = -23
b[0][0][0][1] = -21
b[0][0][0][2] = -19
b[0][0][0][3] = -17
b[0][0][1][0] = -15
b[0][0][1][1] = -13
b[0][0][1][2] = -11
b[0][0][1][3] = -9
b[0][0][2][0] = -7
b[0][0][2][1] = -5
b[0][0][2][2] = -3
b[0][0][2][3] = -1
b[0][1][0][0] = 1
b[0][1][0][1] = 3
b[0][1][0][2] = 5
b[0][1][0][3] = 7
b[0][1][1][0] = 9
b[0][1][1][1] = 11
b[0][1][1][2] = 13
b[0][1][1][3] = 15
b[0][1][2][0] = 17
b[0][1][2][1] = 19
b[0][1][2][2] = 21
b[0][1][2][3] = 23

```

可见，BlobProto 对象实现了磁盘、内存之间的数据通信。这对于保存、载入训练好的模型权值非常实用。

8.1.2 数据结构描述

打开 `src/caffe/proto/caffe.proto`，首先映入眼帘的便是与 Blob 相关的描述，可见该数据结构的重要性，是其他大部分数据结构的依赖项。

```
// 该结构描述了 Blob 的形状信息
message BlobShape {
    repeated int64 dim = 1 [packed = true];    //只包括若干 int64 类型值,分别表示 Blob 每个维度的
    大小。packed 表示这些值在内存中紧密排布,没有空洞
}

// 该结构描述 Blob 在磁盘中序列化后的形态
message BlobProto {
    optional BlobShape shape = 7;                // 可选, 包括一个 BlobShape 对象
    repeated float data = 5 [packed = true]; // 包括若干浮点元素, 存储数据或权值, 元素数目由 shape
    或 (num, channels, height, width) 确定, 这些元素在内存中紧密排布
    repeated float diff = 6 [packed = true]; // 包括若干浮点元素, 用于存储增量信息, 维度与 data
    数组一致
    repeated double double_data = 8 [packed = true];    // 与 data 并列, 只是类型为 double
    repeated double double_diff = 9 [packed = true];    // 与 diff 并列, 只是类型为 double

    // 以下为可选的维度信息, 新版本 Caffe 推荐使用 shape, 而不再用后面的值
    optional int32 num = 1 [default = 0];
    optional int32 channels = 2 [default = 0];
    optional int32 height = 3 [default = 0];
    optional int32 width = 4 [default = 0];
}
```

按照 C/C++ 的思路, 我们也可以直接声明上述 BlobShape、BlobProto 为结构体。为什么一定要用 **ProtoBuffer** 这种格式呢? 简单来说, 结构体存在一些使用不方便的地方。首先, 结构体的序列化/反序列化操作需要额外的编程实现, 难以做到接口标准化; 其次, 结构体中包含变长数据(一般用指向某个内存地址的指针)时, 需要更加细致的工作保证数据完整性。而 **ProtoBuffer** 将编程最容易出问题的地方加以隐藏, 让机器自动处理, 提高了程序健壮性。更多的 **ProtoBuffer** 描述规则请参阅参考资料[2]。

8.1.3 Blob 是怎样炼成的

Blob 是一个模板类, 声明在 include/caffe/blob.hpp 中, 封装了 SyncedMemory 类, 作为基本计算单元服务 Layer、Net、Solver 等:

```
#include .....
#include "caffe/proto/caffe.pb.h"
// 由 protoc 生成的头文件, 声明了 BlobProto、BlobShape 等遵循
// caffe.proto 协议的数据结构
#include "caffe/syncedmem.hpp"                // CPU/GPU 共享内存类, 用于数据同步
#include "caffe/util/math_functions.hpp"      // 数学计算函数
www.aibbt.com 让未来触手可及
```

```

const int kMaxBlobAxes = 32; // Blob 最大维数目

template <typename Dtype>
class Blob { // 类声明
public:
    // 默认构造函数
    Blob() : data_(), diff_(), count_(0), capacity_(0) {}
    // 显式构造函数，避免隐式数据类型转换
    explicit Blob(const vector<int>& shape);
    // 变形函数，根据输入参数重新设置当前 Blob 形状，必要时重新分配内存
    void Reshape(const vector<int>& shape);
    void Reshape(const BlobShape& shape);
    void ReshapeLike(const Blob& other);
    // 得到 Blob 形状字符串用于打印 log，见 Caffe 运行 log，类似
    // "Top shape: 100 1 28 28 (78400)"
    inline string shape_string() const {
        ostringstream stream;
        for (int i = 0; i < shape_.size(); ++i) {
            stream << shape_[i] << " ";
        }
        stream << "(" << count_ << ")";
        return stream.str();
    }
    // 返回 Blob 形状
    inline const vector<int>& shape() const { return shape_; }
    // 返回某一维度的尺寸
    inline int shape(int index) const {
        return shape_[CanonicalAxisIndex(index)];
    }
    // 返回维度数目
    inline int num_axes() const { return shape_.size(); }
    // 返回 Blob 中元素总数
    inline int count() const { return count_; }
    // 返回 Blob 中某几维子集的元素总数
    inline int count(int start_axis, int end_axis) const {
        CHECK_LE(start_axis, end_axis); // 保证 start_axis <= end_axis
        CHECK_GE(start_axis, 0); // 保证 start_axis >= 0
        CHECK_GE(end_axis, 0); // 保证 end_axis >= 0
        CHECK_LE(start_axis, num_axes()); // 保证 start_axis <= 总的维度数目
        CHECK_LE(end_axis, num_axes()); // 保证 end_axis <= 总的维度数目
    }
};

```

```

    int count = 1;
    for (int i = start_axis; i < end_axis; ++i) {
        count *= shape(i);
    }
    return count;
}

// 计算从某一维度开始的元素总数
inline int count(int start_axis) const {
    return count(start_axis, num_axes());
}

// 转换坐标轴索引[-N,N)为普通索引[0,N)
inline int CanonicalAxisIndex(int axis_index) const {
    CHECK_GE(axis_index, -num_axes()) // 保证 axis_index >= -num_axes()
    << "axis " << axis_index << " out of range for " << num_axes()
    << "-D Blob with shape " << shape_string();
    CHECK_LT(axis_index, num_axes()) // 保证 axis_index < num_axes()
    << "axis " << axis_index << " out of range for " << num_axes()
    << "-D Blob with shape " << shape_string();
    if (axis_index < 0) {
        return axis_index + num_axes(); // 负索引表示从后向前访问, -1 表示最后一个元素, 普通索引
        值为 N-1; 同理, -2 => N-2, -3 => N-3, ...
    }
    return axis_index;
}

// 获取形状某一维的尺寸
inline int num() const { return LegacyShape(0); }
inline int channels() const { return LegacyShape(1); }
inline int height() const { return LegacyShape(2); }
inline int width() const { return LegacyShape(3); }
inline int LegacyShape(int index) const {
    CHECK_LE(num_axes(), 4)
    << "Cannot use legacy accessors on Blobs with > 4 axes.";
    CHECK_LT(index, 4);
    CHECK_GE(index, -4);
    if (index >= num_axes() || index < -num_axes()) {
        return 1;
    }
    return shape(index);
}

```

```

// 下面这几个函数都是计算偏移量的
inline int offset(const int n, const int c = 0, const int h = 0,
    const int w = 0) const {
    CHECK_GE(n, 0);
    CHECK_LE(n, num());
    CHECK_GE(channels(), 0);
    CHECK_LE(c, channels());
    CHECK_GE(height(), 0);
    CHECK_LE(h, height());
    CHECK_GE(width(), 0);
    CHECK_LE(w, width());
    return ((n * channels() + c) * height() + h) * width() + w;
}

inline int offset(const vector<int>& indices) const {
    CHECK_LE(indices.size(), num_axes());
    int offset = 0;
    for (int i = 0; i < num_axes(); ++i) {
        offset *= shape(i);
        if (indices.size() > i) {
            CHECK_GE(indices[i], 0);
            CHECK_LT(indices[i], shape(i));
            offset += indices[i];
        }
    }
    return offset;
}

// 按值拷贝 Blob 到当前 Blob
void CopyFrom(const Blob<Dtype>& source, bool copy_diff = false,
    bool reshape = false);

// 这几个函数都是存取器 (getter/setter)
inline Dtype data_at(const int n, const int c, const int h,
    const int w) const {
    return cpu_data()[offset(n, c, h, w)];
}

inline Dtype diff_at(const int n, const int c, const int h,
    const int w) const {
    return cpu_diff()[offset(n, c, h, w)];
}

inline Dtype data_at(const vector<int>& index) const {
    return cpu_data()[offset(index)];
}

```



```

}
inline Dtype diff_at(const vector<int>& index) const {
    return cpu_diff()[offset(index)];
}
inline const shared_ptr<SyncedMemory>& data() const {
    CHECK(data_);
    return data_;
}
inline const shared_ptr<SyncedMemory>& diff() const {
    CHECK(diff_);
    return diff_;
}
// 只读访问 cpu data
const Dtype* cpu_data() const;
// 设置 cpu data
void set_cpu_data(Dtype* data);
// 只读访问 gpu data
const Dtype* gpu_data() const;
// 只读访问 cpu diff
const Dtype* cpu_diff() const;
// 只读访问 gpu diff
const Dtype* gpu_diff() const;
// 读写访问 cpu data
Dtype* mutable_cpu_data();
// 读写访问 gpu data
Dtype* mutable_gpu_data();
// 读写访问 cpu diff
Dtype* mutable_cpu_diff();
// 读写访问 gpu diff
Dtype* mutable_gpu_diff();
void Update(); // Blob 更新运算, 可简单理解为 data 与 diff 的 merge 过程
// 反序列化函数, 从 BlobProto 中恢复一个 Blob 对象
void FromProto(const BlobProto& proto, bool reshape = true);
// 序列化函数, 将内存中的 Blob 对象保存到 BlobProto 中
void ToProto(BlobProto* proto, bool write_diff = false) const;
// 计算 data 的 L1-范数
Dtype asum_data() const;
// 计算 diff 的 L1-范数
Dtype asum_diff() const;

```

```

// 计算 data 的 L2-范数
Dtype sumsq_data() const;
// 计算 diff 的 L2-范数
Dtype sumsq_diff() const;
// data 乘以一个标量
void scale_data(Dtype scale_factor);
// diff 乘以一个标量
void scale_diff(Dtype scale_factor);
// 共享另一个 Blob 的 data_
void ShareData(const Blob& other);

void ShareDiff(const Blob& other);    // 共享另一个 Blob 的 diff_
bool ShapeEquals(const BlobProto& other);
protected:
    shared_ptr<SyncedMemory> data_;    // 存放指向 data 的指针
    shared_ptr<SyncedMemory> diff_;    // 存放指向 diff 的指针
    vector<int> shape_;                // 形状信息
    int count_;                        // 存放有效元素数目信息
    int capacity_;                     // 存放 Blob 容器的容量信息
    DISABLE_COPY_AND_ASSIGN(Blob);    // 禁用拷贝构造函数、赋值运算符重载
}; // class Blob

```

注意到 Caffe 类中成员变量名都带有后缀 “_”，这样在函数实现中容易区分临时变量和类成员变量。

打开 include/caffe/syncedmem.hpp，查看该类的用法：

```

#ifndef CAFFE_SYNCEDMEM_HPP_
#define CAFFE_SYNCEDMEM_HPP_

#include <cstdlib>

#include "caffe/common.hpp"
#include "caffe/util/math_functions.hpp"

namespace caffe {

// 如果在 GPU 模式，且 CUDA 使能，那么主机内存会以页锁定内存方式分配（使用 cudaMallocHost() 函数。对于单 GPU 的性能提升不明显，但多 GPU 会非常明显
inline void CaffeMallocHost(void** ptr, size_t size) {

```

```

#ifdef CPU_ONLY
    if (Caffe::mode() == Caffe::GPU) {
        CUDA_CHECK(cudaMallocHost(ptr, size));
        return;
    }
#endif

*ptr = malloc(size);
CHECK(*ptr) << "host allocation of size " << size << " failed";
}

// 与 CaffeMallocHost 对应
inline void CaffeFreeHost(void* ptr) {
#ifdef CPU_ONLY
    if (Caffe::mode() == Caffe::GPU) {
        CUDA_CHECK(cudaFreeHost(ptr));
        return;
    }
#endif
    free(ptr);
}

// 该类负责存储分配以及主机和设备间同步
class SyncedMemory {
public:
    // 构造函数
    SyncedMemory()
        : cpu_ptr_(NULL), gpu_ptr_(NULL), size_(0), head_(UNINITIALIZED),
          own_cpu_data_(false), own_gpu_data_(false), gpu_device_(-1) {}
    // 显式构造函数
    explicit SyncedMemory(size_t size)
        : cpu_ptr_(NULL), gpu_ptr_(NULL), size_(size), head_(UNINITIALIZED),
          own_cpu_data_(false), own_gpu_data_(false), gpu_device_(-1) {}
    // 析构函数
    ~SyncedMemory();
    // Getters/Setters
    const void* cpu_data();           // 只读获取 cpu data
    void set_cpu_data(void* data);    // 设置 cpu data
    const void* gpu_data();           // 只读获取 gpu data
    void set_gpu_data(void* data);    // 设置 gpu data

```

```

void* mutable_cpu_data();          // 读写获取 cpu data
void* mutable_gpu_data();          // 读写获取 gpu data
// 状态机变量，表示 4 种状态：未初始化、CPU 数据有效、GPU 数据有效、已同步
enum SyncedHead { UNINITIALIZED, HEAD_AT_CPU, HEAD_AT_GPU, SYNCED };
// 获得当前状态机变量值
SyncedHead head() { return head_; }
// 获得当前存储空间尺寸
size_t size() { return size_; }

#ifdef CPU_ONLY
    void async_gpu_push(const cudaStream_t& stream);
#endif

private:
    void to_cpu();          // 数据同步至 CPU
    void to_gpu();          // 数据同步至 GPU
    void* cpu_ptr_;         // 位于 CPU 的数据指针
    void* gpu_ptr_;         // 位于 GPU 的数据指针
    size_t size_;           // 存储空间大小
    SyncedHead head_;       // 状态机变量
    bool own_cpu_data_;     // 标志是否拥有 CPU 数据所有权（否，即从别的对象共享）
    bool own_gpu_data_;     // 标志是否拥有 GPU 数据所有权
    int gpu_device_;        // GPU 设备号

    DISABLE_COPY_AND_ASSIGN(SyncedMemory);
}; // class SyncedMemory

} // namespace caffe

#endif // CAFFE_SYNCEDMEM_HPP_

```

Blob 类实现的源码位于 `src/caffe/blob.cpp` 中，内容如下：

```

#include <climits>
#include <vector>

#include "caffe/blob.hpp"
#include "caffe/common.hpp"
#include "caffe/syncedmem.hpp"
#include "caffe/util/math_functions.hpp"

```

```

namespace caffe {
// 变维函数，将(num, channels, height, width) 参数转换为 vector<int>，然后调用重载的变维函数 void
Blob<Dtype>::Reshape(const vector<int>& shape)
template <typename Dtype>
void Blob<Dtype>::Reshape(const int num, const int channels, const int height,
    const int width) {
    vector<int> shape(4);
    shape[0] = num;
    shape[1] = channels;
    shape[2] = height;
    shape[3] = width;
    Reshape(shape);
}
// 真正变维函数
template <typename Dtype>
void Blob<Dtype>::Reshape(const vector<int>& shape) {
    CHECK_LE(shape.size(), kMaxBlobAxes); // 保证 vector 维度<=kMaxBlobAxes
    count_ = 1; // 用于计算元素总数 = num * channels * height * width
    shape_.resize(shape.size()); // 成员变量维度也被重置
    for (int i = 0; i < shape.size(); ++i) {
        CHECK_GE(shape[i], 0); // 保证每维度尺寸都>= 0
        // 保证 count_ 不溢出
        CHECK_LE(shape[i], INT_MAX / count_) << "blob size exceeds INT_MAX";
        count_ *= shape[i]; // count_ 累乘
        shape_[i] = shape[i]; // 为成员变量赋值
    }
    if (count_ > capacity_) { // 如果新的 count_ 大于当前已分配空间容量
        capacity_ = count_; // 扩容，重新分配 data_ 和 diff_ 空间
        data_.reset(new SyncedMemory(capacity_ * sizeof(Dtype)));
        diff_.reset(new SyncedMemory(capacity_ * sizeof(Dtype)));
    }
}
// void Blob<Dtype>::Reshape(const BlobShape& shape) 和
// void Blob<Dtype>::ReshapeLike(const Blob<Dtype>& other) 类似，略
// 构造函数
template <typename Dtype>
Blob<Dtype>::Blob(const int num, const int channels, const int height,
    const int width)
// 调用 Reshape 之前必须初始化 capacity_，否则会导致不可预期结果

```

```

: capacity_(0) {
    Reshape(num, channels, height, width);
}
// 只读获得 cpu data 指针
template <typename Dtype>
const Dtype* Blob<Dtype>::cpu_data() const {
    CHECK(data_); // 保证 data_ 不为 NULL
    return (const Dtype*)data_>cpu_data();
}
// 修改 cpu data 指针
template <typename Dtype>
void Blob<Dtype>::set_cpu_data(Dtype* data) {
    CHECK(data); // 保证 data 不为 NULL
    data_>set_cpu_data(data); // 设置成员变量值为传入参数值
}
// 只读获得 gpu data 指针
template <typename Dtype>
const Dtype* Blob<Dtype>::gpu_data() const {
    CHECK(data_); // 保证 data_ 不为 NULL
    return (const Dtype*)data_>gpu_data();
}
// 只读获得 cpu diff 指针
template <typename Dtype>
const Dtype* Blob<Dtype>::cpu_diff() const {
    CHECK(diff_); // 保证 diff_ 不为 NULL
    return (const Dtype*)diff_>cpu_data();
}
// 只读获得 gpu diff 指针
template <typename Dtype>
const Dtype* Blob<Dtype>::gpu_diff() const {
    CHECK(diff_); // 保证 diff_ 不为 NULL
    return (const Dtype*)diff_>gpu_data();
}
// 读写访问 cpu data 指针
template <typename Dtype>
Dtype* Blob<Dtype>::mutable_cpu_data() {
    CHECK(data_);
    return static_cast<Dtype*>(data_>mutable_cpu_data());
}

```

```

// 读写访问 gpu data 指针
template <typename Dtype>
Dtype* Blob<Dtype>::mutable_gpu_data() {
    CHECK(data_);
    return static_cast<Dtype*>(data_>mutable_gpu_data());
}

// 读写访问 cpu diff 指针
template <typename Dtype>
Dtype* Blob<Dtype>::mutable_cpu_diff() {
    CHECK(diff_);
    return static_cast<Dtype*>(diff_>mutable_cpu_data());
}

// 读写访问 gpu diff 指针
template <typename Dtype>
Dtype* Blob<Dtype>::mutable_gpu_diff() {
    CHECK(diff_);
    return static_cast<Dtype*>(diff_>mutable_gpu_data());
}

// 共享另一个 Blob 的 data 指针
template <typename Dtype>
void Blob<Dtype>::ShareData(const Blob& other) {
    CHECK_EQ(count_, other.count());
    data_ = other.data();
}

// 共享另一个 Blob 的 diff 指针
template <typename Dtype>
void Blob<Dtype>::ShareDiff(const Blob& other) {
    CHECK_EQ(count_, other.count());
    diff_ = other.diff();
}

// Update() 函数用于网络参数 Blob 的更新。其中 int 和 unsigned int 类型处理并未实现
template <> void Blob<unsigned int>::Update() { NOT_IMPLEMENTED; }
template <> void Blob<int>::Update() { NOT_IMPLEMENTED; }
template <typename Dtype>
void Blob<Dtype>::Update() {
    // data 在哪里，就在哪里更新
    switch (data_>head()) {
        case SyncedMemory::HEAD_AT_CPU: // data 位于 CPU 端

```

```

// 执行CPU上的计算, data_[i] = data_[i] - diff_[i], i = 0, 1, 2, ..., count_-1
caffe_axpy<Dtype>(count_, Dtype(-1),
    static_cast<const Dtype*>(diff_->cpu_data()),
    static_cast<Dtype*>(data_->mutable_cpu_data()));
break;
case SyncedMemory::HEAD_AT_GPU: // data 位于GPU端, 或者CPU/GPU已同步
case SyncedMemory::SYNCED:
#ifdef CPU_ONLY
    // 执行GPU上的计算, data_[i] = data_[i] - diff_[i], i = 0, 1, 2, ..., count_-1
    caffe_gpu_axpy<Dtype>(count_, Dtype(-1),
        static_cast<const Dtype*>(diff_->gpu_data()),
        static_cast<Dtype*>(data_->mutable_gpu_data()));
#else
    NO_GPU; // 编译时打开了CPU_ONLY选项, 那么GPU模式禁用
#endif
    break;
default:
    LOG(FATAL) << "Syncedmem not initialized.";
}
}

// 计算data_的L1-范数
template <typename Dtype>
Dtype Blob<Dtype>::asum_data() const {
    if (!data_) { return 0; }
    switch (data_->head()) {
    case SyncedMemory::HEAD_AT_CPU:
        return caffe_cpu_asum(count_, cpu_data()); // 执行CPU上的asum计算
    case SyncedMemory::HEAD_AT_GPU:
    case SyncedMemory::SYNCED:
#ifdef CPU_ONLY
    {
        Dtype asum;
        caffe_gpu_asum(count_, gpu_data(), &asum); // 执行GPU上的asum计算
        return asum;
    }
#else
    NO_GPU;
#endif
    }
}

```



```

case SyncedMemory::UNINITIALIZED:
    return 0;
default:
    LOG(FATAL) << "Unknown SyncedMemory head state: " << data_>->head();
}
return 0;
}

// 计算 data_ 的 L2-范数
template <typename Dtype>
Dtype Blob<Dtype>::sumsq_data() const {
    Dtype sumsq;
    const Dtype* data;
    if (!data_) { return 0; }
    switch (data_>head()) {
    case SyncedMemory::HEAD_AT_CPU:
        data = cpu_data();
        sumsq = caffe_cpu_dot(count_, data, data); // 执行 CPU 上的 dot 计算
        break;
    case SyncedMemory::HEAD_AT_GPU:
    case SyncedMemory::SYNCED:
#ifdef CPU_ONLY
        data = gpu_data();
        caffe_gpu_dot(count_, data, data, &sumsq); // 执行 GPU 上的 dot 计算
#else
        NO_GPU;
#endif
        break;
    case SyncedMemory::UNINITIALIZED:
        return 0;
    default:
        LOG(FATAL) << "Unknown SyncedMemory head state: " << data_>->head();
    }
    return sumsq;
}

// 对 data_ 进行幅度缩放
template <typename Dtype>
void Blob<Dtype>::scale_data(Dtype scale_factor) {
    Dtype* data;
    if (!data_) { return; }

```

```

switch (data_>head()) {
case SyncedMemory::HEAD_AT_CPU:
    data = mutable_cpu_data(); // 执行 CPU 上的计算
    caffe_scal(count_, scale_factor, data); // data[i] = data[i] * scale_factor, i = 0,
1, 2, ..., count_-1
    return;
case SyncedMemory::HEAD_AT_GPU:
case SyncedMemory::SYNCED:
#ifdef CPU_ONLY
    data = mutable_gpu_data(); // 执行 GPU 上的计算
    caffe_gpu_scal(count_, scale_factor, data);
    return;
#else
    NO_GPU;
#endif
case SyncedMemory::UNINITIALIZED:
    return;
default:
    LOG(FATAL) << "Unknown SyncedMemory head state: " << data_>head();
}
}

// 判断形状是否相同
template <typename Dtype>
bool Blob<Dtype>::ShapeEquals(const BlobProto& other) {
    if (other.has_num() || other.has_channels() ||
        other.has_height() || other.has_width()) {
        // 输入的维度若使用过时的维度信息(num, channels, height, width), 则需要转换为新的 vector 参
        // 数代码使用了 C++ 中的“懒”逻辑
        return shape_.size() <= 4 &&
            LegacyShape(-4) == other.num() &&
            LegacyShape(-3) == other.channels() &&
            LegacyShape(-2) == other.height() &&
            LegacyShape(-1) == other.width();
    }
}

// 直接对比
vector<int> other_shape(other.shape().dim_size());
for (int i = 0; i < other.shape().dim_size(); ++i) {
    other_shape[i] = other.shape().dim(i);
}

return shape_ == other_shape;

```

```

}
// 从另一个 Blob 对象拷贝 data (可选 diff), 必要时进行变维
template <typename Dtype>
void Blob<Dtype>::CopyFrom(const Blob& source, bool copy_diff, bool reshape) {
    if (source.count() != count_ || source.shape() != shape_) {
        if (reshape) {
            ReshapeLike(source); // 要求变维, 那就照做
        } else { // 两个 Blob 形状不同, 硬要拷贝, “臣妾做不到啊”
            LOG(FATAL) << "Trying to copy blobs of different sizes.";
        }
    }
    switch (Caffe::mode()) {
    case Caffe::GPU: // 如果使用 GPU 模式, 就用 gpu 的方法
        if (copy_diff) {
            caffe_copy(count_, source.gpu_diff(),
                static_cast<Dtype*>(diff->mutable_gpu_data())); // diff -> diff
        } else {
            caffe_copy(count_, source.gpu_data(),
                static_cast<Dtype*>(data->mutable_gpu_data())); // data -> data
        }
        break;
    case Caffe::CPU: // 如果使用 CPU 模式, 就用 cpu 的方法
        if (copy_diff) {
            caffe_copy(count_, source.cpu_diff(),
                static_cast<Dtype*>(diff->mutable_cpu_data()));
        } else {
            caffe_copy(count_, source.cpu_data(),
                static_cast<Dtype*>(data->mutable_cpu_data()));
        }
        break;
    default:
        LOG(FATAL) << "Unknown caffe mode.";
    }
}

// 从 BlobProto 中加载一个 Blob, 适用于从磁盘载入之前导出的 Blob
template <typename Dtype>
void Blob<Dtype>::FromProto(const BlobProto& proto, bool reshape) {
    if (reshape) { // 从 BlobProto 对象中获得所需各个维度信息
        vector<int> shape;
    }
}

```

```

if (proto.has_num() || proto.has_channels() ||
    proto.has_height() || proto.has_width()) {
    // 过时的维度信息 (num, channels, height, width)
    shape.resize(4);
    shape[0] = proto.num();
    shape[1] = proto.channels();
    shape[2] = proto.height();
    shape[3] = proto.width();
} else {
    shape.resize(proto.shape().dim_size());
    for (int i = 0; i < proto.shape().dim_size(); ++i) {
        shape[i] = proto.shape().dim(i);
    }
}

Reshape(shape); // Blob 按照维度信息进行变维
} else {
    CHECK(ShapeEquals(proto)) << "shape mismatch (reshape not set)";
}

// 加载数据
Dtype* data_vec = mutable_cpu_data();
if (proto.double_data_size() > 0) { // 如果之前保存的是 double 类型 data
    CHECK_EQ(count_, proto.double_data_size());
    for (int i = 0; i < count_; ++i) {
        data_vec[i] = proto.double_data(i); // 加载 double data
    }
} else {
    CHECK_EQ(count_, proto.data_size());
    for (int i = 0; i < count_; ++i) {
        data_vec[i] = proto.data(i); // 否则加载 float data
    }
}

if (proto.double_diff_size() > 0) { // 如果之前保存的是 double 类型 diff
    CHECK_EQ(count_, proto.double_diff_size());
    Dtype* diff_vec = mutable_cpu_diff();
    for (int i = 0; i < count_; ++i) {
        diff_vec[i] = proto.double_diff(i); // 加载 double diff
    }
} else if (proto.diff_size() > 0) {
    CHECK_EQ(count_, proto.diff_size());

```

```

Dtype* diff_vec = mutable_cpu_diff();
for (int i = 0; i < count_; ++i) {
    diff_vec[i] = proto.diff(i); // 否则加载 float diff
}
}

// 将 Blob 中的 data (可选 diff) 导出到 BlobProto 结构体, 便于存储到磁盘文件中
template <>
void Blob<float>::ToProto(BlobProto* proto, bool write_diff) const {
    proto->clear_shape(); // 重置 proto 的维度, 保证与 Blob 相同
    for (int i = 0; i < shape_.size(); ++i) {
        proto->mutable_shape()->add_dim(shape_[i]);
    }
    proto->clear_data(); // 清除 data
    proto->clear_diff(); // 清除 diff
    const float* data_vec = cpu_data(); // 将 data 导出到 proto
    for (int i = 0; i < count_; ++i) {
        proto->add_data(data_vec[i]);
    }
    if (write_diff) { // 若有 write_diff 的需求
        const float* diff_vec = cpu_diff(); // 将 diff 导出到 proto
        for (int i = 0; i < count_; ++i) {
            proto->add_diff(diff_vec[i]);
        }
    }
}

INSTANTIATE_CLASS(Blob); // 实例化 Blob 类模板 (float, double)
template class Blob<int>;
template class Blob<unsigned int>;

} // namespace caffe

```

至此, 我们看完了 Caffe 的砖块是如何烧制的。下面准备开始搬砖!

8.2 Layer

Layer 是 Caffe 的基本计算单元, 至少有一个输入 Blob (Bottom Blob) 和一个输出 Blob (Top

Blob)，部分 Layer 带有权值（Weight）和偏置项（Bias），有两个运算方向：前向传播（Forward）和反向传播（Backward），其中前向传播计算会对输入 Blob 进行某种处理（有权值和偏置项的 Layer 会利用这些对输入进行处理），得到输出 Blob；而反向传播计算则对输出 Blob 的 diff 进行某种处理，得到输入 Blob 的 diff（有权值和偏置项的 Layer 可能也会计算权值 Blob、偏置项 Blob 的 diff）。

8.2.1 数据结构描述

```
// 注意：如果你增加了一个新的 LayerParameter 域，一定记得更新下一个可用 ID
//下一个 ID 为 137，最近更新：reduction_param

message LayerParameter {
    optional string name = 1;      // Layer 名称
    optional string type = 2;      // Layer 类型
    repeated string bottom = 3;    // 输入 Blob (Bottom Blob) 的名称
    repeated string top = 4;       // 输出 Blob (Top Blob) 的名称
    optional Phase phase = 10;     // 当前阶段 (TRAIN 或 TEST)

    // 为每个 Top Blob 分配对损失函数的权重，每个 Layer 都有默认值，要么为 0，表示不参与目标函数计算；要么
    // 为 1，表示参与损失函数计算
    repeated float loss_weight = 5;

    repeated ParamSpec param = 6; // 指定训练参数（例如相对全局学习常数的缩放因子，以及用于权值共享
    // 的名称或其他设置）
    repeated BlobProto blobs = 7; // 承载了该层数值参数的 Blob
    repeated bool propagate_down = 11; // 是否对 Bottom Blob 进行反向传播过程。该字段的维度应与
    // Bottom Blob 个数一致

    // 控制某个层在某个时刻是否包含在网络中，基于当前 NetState。你可以为 include 或 exclude（不要同时）
    // 指定非零值。如果没有任何规则，那么该层一直包含在网络中；如果当前 NetState 满足了任何一个指定规则，那么
    // 该层会被包含或排斥
    repeated NetStateRule include = 8;
    repeated NetStateRule exclude = 9;

    optional TransformationParameter transform_param = 100; // 数据预处理参数
    optional LossParameter loss_param = 101; // 所有损失层共享的参数

    // 特定类型层的参数。注意一些层实现时可能有多于一种的计算引擎，这些层包括一个引擎类型和引擎参数来选择
    // 实现。默认引擎是在编译阶段由引擎开关设置的
    optional AccuracyParameter accuracy_param = 102;
    optional ArgMaxParameter argmax_param = 103;
    optional ConcatParameter concat_param = 104;
    optional ContrastiveLossParameter contrastive_loss_param = 105;
    optional ConvolutionParameter convolution_param = 106;
    optional DataParameter data_param = 107;
```

```

optional DropoutParameter dropout_param = 108;
optional DummyDataParameter dummy_data_param = 109;
optional EltwiseParameter eltwise_param = 110;
optional ExpParameter exp_param = 111;
optional FlattenParameter flatten_param = 135;
optional HDF5DataParameter hdf5_data_param = 112;
optional HDF5OutputParameter hdf5_output_param = 113;
optional HingeLossParameter hinge_loss_param = 114;
optional ImageDataParameter image_data_param = 115;
optional InfogainLossParameter infogain_loss_param = 116;
optional InnerProductParameter inner_product_param = 117;
optional LogParameter log_param = 134;
optional LRNParameter lrn_param = 118;
optional MemoryDataParameter memory_data_param = 119;
optional MVNParameter mvn_param = 120;
optional PoolingParameter pooling_param = 121;
optional PowerParameter power_param = 122;
optional PReLUParameter prelu_param = 131;
optional PythonParameter python_param = 130;
optional ReductionParameter reduction_param = 136;
optional ReLUParameter relu_param = 123;
optional ReshapeParameter reshape_param = 133;
optional SigmoidParameter sigmoid_param = 124;
optional SoftmaxParameter softmax_param = 125;
optional SPPParameter spp_param = 132;
optional SliceParameter slice_param = 126;
optional TanHParameter tanh_param = 127;
optional ThresholdParameter threshold_param = 128;
optional WindowDataParameter window_data_param = 129;
}

```

8.2.2 Layer 是怎样建成的

Layer 头文件位于 include/caffe/layer.hpp 中，内容如下：

```

template <typename Dtype>
class Layer {
public:
    // 显式构造函数，从 LayerParameter 对象中加载配置
    explicit Layer(const LayerParameter& param)

```

```

: layer_param_(param), is_shared_(false) {
    phase_ = param.phase(); // 设置当前阶段 (训练/预测)
    if (layer_param_.blobs_size() > 0) {
        blobs_.resize(layer_param_.blobs_size()); // 按 layer_param_ 设置本身 Blob 对象个数,
        并依次将每个 Blob 对象尺寸调整为与 layer_param_ 中的 Blob 尺寸一致
        for (int i = 0; i < layer_param_.blobs_size(); ++i) {
            blobs_[i].reset(new Blob<Dtype>());
            blobs_[i]->FromProto(layer_param_.blobs(i));
        }
    }
}

virtual ~Layer() {} // 虚析构函数
// 配置函数, 实现常用层配置接口, 不可被覆盖
void SetUp(const vector<Blob<Dtype>*>& bottom,
           const vector<Blob<Dtype>*>& top) {
    InitMutex();
    CheckBlobCounts(bottom, top); // 检查 Blob
    LayerSetUp(bottom, top);      // 与层类型相关的配置过程
    Reshape(bottom, top);         // 对 Top Blob 变形
    SetLossWeights(top);          // 设置损失权重因子 Blob
}

// 层配置 (虚) 函数, 做特定类型层相关的配置, 由该类型层自己实现
virtual void LayerSetUp(const vector<Blob<Dtype>*>& bottom,
                        const vector<Blob<Dtype>*>& top) {}

// 在数据并行中, 一个 Layer 是否被多个 Net 共享。在默认情况下, 只有数据读取层 (DataLayer) 可被多个
Net 共享, 其他层则不能
virtual inline bool ShareInParallel() const { return false; } // 默认与否
// 返回该层实际上是否被其他 Net 共享。当 ShareInParallel() 返回 true, 以及多个 GPU 被使用, 网络处于
训练阶段时, 该函数返回 true
inline bool IsShared() const { return is_shared_; }
// 设置该层实际上是否被其他 Net 共享, 条件同上
inline void SetShared(bool is_shared) {
    CHECK(ShareInParallel() || !is_shared)
<< type() << "Layer does not support sharing.";
    is_shared_ = is_shared;
}

// 变形 (纯虚) 函数, 修改 Top Blob 以及内部 Blob 缓冲区的形状

```



```

virtual void Reshape(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) = 0;
// 前向传播函数, 给定 Bottom Blob, 计算 Top Blob 和 loss, 返回值为当前层 loss
// 该函数会调用相应设备包装函数, 如 Forward_cpu 或 Forward_gpu 来实现真正的计算过程。如果该层有任
// 意非零 loss_weights 参数, 那么包装函数会计算并返回 loss
// 派生类应该实现 Forward_cpu 和 Forward_gpu (可选)
inline Dtype Forward(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top);
// 反向传播函数, 给定 Top Blob 误差梯度, 计算 Bottom Blob 误差梯度
// 参数说明:
// top—Top Blob, 其 diff 域包含来自上一层的误差梯度
// propagate_down—多路开关, 与 Bottom Blob 矢量维度相同, 每个值表示是否将误差梯度传递到对应的
Bottom Blob
// bottom—Bottom Blob, 其 diff 域需要由该函数计算得到
// 该函数会调用相应设备包装函数, 如 Backward_cpu 或 Backward_gpu 来实现真正的计算过程, 由派生类负
责实现
inline void Backward(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom);
// 返回 Layer 内部可训练的权值、偏置项 Blob 向量
vector<shared_ptr<Blob<Dtype>>>& blobs() {
    return blobs_;
}
// 返回 Layer 初始化参数 (由 ProtoBuffer 提供)
const LayerParameter& layer_param() const { return layer_param_; }
// 将 Layer 初始化参数写入 ProtoBuffer 缓冲区
virtual void ToProto(LayerParameter* param, bool write_diff = false);
// 返回与某个 Top Blob 相关的标量 loss 值
inline Dtype loss(const int top_index) const {
    return (loss_.size() > top_index) ? loss_[top_index] : Dtype(0);
}
// 设置与某个 Top Blob 相关的标量 loss 值
inline void set_loss(const int top_index, const Dtype value) {
    if (loss_.size() <= top_index) {
        loss_.resize(top_index + 1, Dtype(0));
    }
    loss_[top_index] = value;
}

// 返回层类型字符串, 便于识别, 由派生类负责实现

```

```

virtual inline const char* type() const { return ""; }
// 返回该 Layer 需要的输入 Blob 数目，-1 表示不关心。由派生类负责实现
virtual inline int ExactNumBottomBlobs() const { return -1; }
virtual inline int MinBottomBlobs() const { return -1; }
virtual inline int MaxBottomBlobs() const { return -1; }
// 返回该 Layer 需要的输出 Blob 数目，-1 表示不关心。由派生类负责实现
virtual inline int ExactNumTopBlobs() const { return -1; }
virtual inline int MinTopBlobs() const { return -1; }
virtual inline int MaxTopBlobs() const { return -1; }
// 返回该 Layer 是否有相同的输入/输出 Blob，由派生类负责实现
virtual inline bool EqualNumBottomTopBlobs() const { return false; }
// 返回是否允许匿名 Top Blob，即由该 Layer 自动创建。若为真，在 Net::Init() 函数中会创建足够多的匿名 Top Blob 来满足该 Layer ExactNumTopBlobs()、MinTopBlobs() 需求
virtual inline bool AutoTopBlobs() const { return false; }

// 返回某些 Bottom Blob 是否允许强制反向传播，如果 AllowForceBackward(i) == false，将会忽略 force_backward 设定
virtual inline bool AllowForceBackward(const int bottom_index) const {
    return true;
}

// 指定该 Layer 是否计算相对权值或偏置项的梯度，具体相对谁由 param_id 指定
inline bool param_propagate_down(const int param_id) {
    return (param_propagate_down_.size() > param_id) ?
        param_propagate_down_[param_id] : false;
}
// 设置该 Layer 是否计算相对权值或偏置项的梯度，具体相对谁由 param_id 指定
inline void set_param_propagate_down(const int param_id, const bool value) {
    if (param_propagate_down_.size() <= param_id) {
        param_propagate_down_.resize(param_id + 1, true);
    }
    param_propagate_down_[param_id] = value;
}

protected:

LayerParameter layer_param_; // 保存 Layer 参数的 ProtoBuffer 对象

Phase phase_; // Layer 当前所处阶段，可选 TRAIN 或 TEST

```

```

vector<shared_ptr<Blob<Dtype>>> blobs_; // Layer 内部权值或偏置项，以 Blob 方式组织

vector<bool> param_propagate_down_; // 标志位，是否计算对应参数的误差梯度
vector<Dtype> loss_; // 标志位，在目标函数中，是否每个 Top Blob 都有非零权重

// 下面 4 个函数，我们会在各个 Layer 派生类中经常看到
virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) = 0;

virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
    // LOG(WARNING) << "Using CPU code as backup.";
    return Forward_cpu(bottom, top);
}

virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) = 0;

virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
    // LOG(WARNING) << "Using CPU code as backup.";
    Backward_cpu(top, propagate_down, bottom);
}

// 校验输入/输出 Blob 数目是否满足 Layer 要求
virtual void CheckBlobCounts(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
    if (ExactNumBottomBlobs() >= 0) {
        CHECK_EQ(ExactNumBottomBlobs(), bottom.size())
        << type() << " Layer takes " << ExactNumBottomBlobs()
        << " bottom blob(s) as input.";
    }
    if (MinBottomBlobs() >= 0) {
        CHECK_LE(MinBottomBlobs(), bottom.size())
        << type() << " Layer takes at least " << MinBottomBlobs()
        << " bottom blob(s) as input.";
    }
}

```

```

    }
    if (MaxBottomBlobs() >= 0) {
        CHECK_GE(MaxBottomBlobs(), bottom.size())
        << type() << " Layer takes at most " << MaxBottomBlobs()
        << " bottom blob(s) as input.";
    }
    if (ExactNumTopBlobs() >= 0) {
        CHECK_EQ(ExactNumTopBlobs(), top.size())
        << type() << " Layer produces " << ExactNumTopBlobs()
        << " top blob(s) as output.";
    }
    if (MinTopBlobs() >= 0) {
        CHECK_LE(MinTopBlobs(), top.size())
        << type() << " Layer produces at least " << MinTopBlobs()
        << " top blob(s) as output.";
    }
    if (MaxTopBlobs() >= 0) {
        CHECK_GE(MaxTopBlobs(), top.size())
        << type() << " Layer produces at most " << MaxTopBlobs()
        << " top blob(s) as output.";
    }
    if (EqualNumBottomTopBlobs()) {
        CHECK_EQ(bottom.size(), top.size())
        << type() << " Layer produces one top blob as output for each "
        << "bottom blob input.";
    }
}

```

// 该函数在 Layer 的 SetUp 函数中被调用，主要目的是初始化与 Top Blob 相关的 loss 权重，放到 Top Blob 的 diff 域，实际由 Forward() 计算 loss 函数

// loss_weight == 0，表示当前层不参与 loss 函数计算，大部分 Layer 属于这一类

// loss_weight == 1，表示当前层参与 loss 函数计算，损失层 (LossLayer) 属于这一类

```
inline void SetLossWeights(const vector<Blob<Dtype>*>& top) {
```

// 从 ProtoBuffer 对象中获得 Layer 参数，这里需要用 loss_weight 参数

```
const int num_loss_weights = layer_param_.loss_weight_size();
```

```
if (num_loss_weights) { // 如果 ProtoBuffer 中存在至少一个 loss_weight 参数
```

// loss_weight 参数个数应当与 Top Blob 数目相同，或者不要 loss_weight 参数

```
CHECK_EQ(top.size(), num_loss_weights) << "loss_weight must be "
```

```
"unspecified or specified once per top blob.";
```

```
// 遍历每个 Top Blob
```

```

for (int top_id = 0; top_id < top.size(); ++top_id) {
    // 从 ProtoBuffer 对象拿到 loss_weight 实际值 (0 或者 1)
    const Dtype loss_weight = layer_param_.loss_weight(top_id);
    // 若为 0, 跳过
    if (loss_weight == Dtype(0)) { continue; }
    // 若不为 0, 则对网络做相关设置
    this->set_loss(top_id, loss_weight); // 本地记录 loss_weight 值
    const int count = top[top_id]->count();
    Dtype* loss_multiplier = top[top_id]->mutable_cpu_diff();
    caffe_set(count, loss_weight, loss_multiplier); // 将 loss_weight 值写入 Top Blob
    // 的 diff 域, 传递到其他需要使用的地方, 实现远程同步
}
}
}

```

private:

```

bool is_shared_; // 标志位, 表明该 Layer 是否被其他 Net 共享

shared_ptr<boost::mutex> forward_mutex; // 如果该 Layer 被共享, 则需要该信号量保证顺序执行 forward
void InitMutex(); // 初始化 forward_mutex_
void Lock(); // 加锁
void Unlock(); // 解锁
DISABLE_COPY_AND_ASSIGN(Layer); // 禁用拷贝构造函数和赋值运算函数
}; // class Layer

```

// 前向传播函数、后向传播函数包装。不需要修改这两个函数

// 使用时只需在派生类中改写 Forward_cpu、Forward_gpu、Backward_cpu、Backward_gpu

template <typename Dtype>

```

inline Dtype Layer<Dtype>::Forward(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {

```

// 加锁, 防止其他 Net 使用

Lock();

Dtype loss = 0;

Reshape(bottom, top);

switch (Caffe::mode()) { // 判断计算设备

case Caffe::CPU: // 在 CPU 上执行 Forward 计算

Forward_cpu(bottom, top); // 调用 CPU 版本的 Forward 函数

// 还没完, 要计算 loss (如果有的话)

for (int top_id = 0; top_id < top.size(); ++top_id) {

```

    if (!this->loss(top_id)) { continue; }
    const int count = top[top_id]->count();
    const Dtype* data = top[top_id]->cpu_data(); // 若为 LossLayer, 则已经通过 Forward 函
    数计算出全局损失函数, 放在 Top Blob data 域
    const Dtype* loss_weights = top[top_id]->cpu_diff(); // 若 loss_weight 不为 0, 则已经
    在 SetLossWeights 函数中将 loss 权重放在 Top Blob diff 域
    loss += caffe_cpu_dot(count, data, loss_weights); // 计算加权后的 loss 之和, 得到标量
    loss 值
}
break;
case Caffe::GPU:
    Forward_gpu(bottom, top);
#ifdef CPU_ONLY
    for (int top_id = 0; top_id < top.size(); ++top_id) {
        if (!this->loss(top_id)) { continue; }
        const int count = top[top_id]->count();
        const Dtype* data = top[top_id]->gpu_data();
        const Dtype* loss_weights = top[top_id]->gpu_diff();
        Dtype blob_loss = 0;
        caffe_gpu_dot(count, data, loss_weights, &blob_loss);
        loss += blob_loss;
    }
#endif
    break;
default:
    LOG(FATAL) << "Unknown caffe mode.";
}
Unlock(); // 解锁, 其他 Net 可以使用
return loss;
}

// 反向传播函数, 直接调用对应设备函数
template <typename Dtype>
inline void Layer<Dtype>::Backward(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
    switch (Caffe::mode()) {
    case Caffe::CPU:
        Backward_cpu(top, propagate_down, bottom);
        break;
    case Caffe::GPU:

```

```

    Backward_gpu(top, propagate_down, bottom);
    break;
default:
    LOG(FATAL) << "Unknown caffe mode.";
}
}

// 将层配置参数序列化为 ProtoBuffer
template <typename Dtype>
void Layer<Dtype>::ToProto(LayerParameter* param, bool write_diff) {
    param->Clear();
    param->CopyFrom(layer_param_);
    param->clear_blobs();
    for (int i = 0; i < blobs_.size(); ++i) { // 权值和偏置项也会保存
        blobs_[i]->ToProto(param->add_blobs(), write_diff);
    }
}
}

```

Layer 源文件位于 src/caffe/layer.cpp 中，内容如下：

```

#include <boost/thread.hpp>
#include "caffe/layer.hpp"

namespace caffe {
// 初始化信号量
template <typename Dtype>
void Layer<Dtype>::InitMutex() {
    forward_mutex_.reset(new boost::mutex());
}

// 加锁
template <typename Dtype>
void Layer<Dtype>::Lock() {
    if (IsShared()) {
        forward_mutex_->lock();
    }
}

// 解锁
template <typename Dtype>
void Layer<Dtype>::Unlock() {
    if (IsShared()) {
        forward_mutex_->unlock();
    }
}
}

```

```

    }
}

INstantiate_Class(Layer);

} // namespace caffe

```

可见 Layer 大部分函数并没有实现，只有虚函数，真正的实现都在派生类中。具体代码可以进一步阅读 `src/caffe/layers/*.cpp`。

在使用 Layer 之前，需要先包含头文件“`#include <caffe/layer.hpp>`”，再通过“`using namespace caffe;`”使用命名空间 `caffe`。如果代码中试图创建 Layer 对象，编译时会报错：

```
error: cannot declare variable 'a' to be of abstract type 'caffe::Layer<float>'
```

这是因为 Layer 类是一个虚基类，不能直接创建对象。如果需要了解更多的关于虚基类的信息，请参阅参考资料[2]。

8.3 Net

Net 在 Caffe 中代表一个完整的 CNN 模型，它包含若干 Layer 实例。前面我们已经在第 5 天内容中看到用 ProtoBuffer 文本文件（`prototxt`）描述的经典网络结构如 LeNet、AlexNet，这些结构反映在 Caffe 代码实现上就是一个 Net 对象。通过本节的学习，读者会发现 Net 是相对 Blob、Layer 更为复杂的设计，需要沉住气，细细阅读。

8.3.1 Net 基本用法

Net 是一张图纸，对应的描述文件为 `*.prototxt`，我们选择 Caffe 自带的 CaffeNet 模型描述文件，位于 `models/bvlc_reference_caffenet/deploy.prototxt`。将该文件拷贝到当前工作目录下。

编写测试代码 `net_demo.cpp` 如下：

```

#include <vector>
#include <iostream>
#include <caffe/net.hpp>
using namespace caffe;
using namespace std;
int main(void)

```



```
{
    std::string proto("deploy.prototxt");
    Net<float> nn(proto, caffe::TEST);
    vector<string> bn = nn.blob_names(); // 获取 Net 中所有 Blob 对象名
    for(int i = 0; i < bn.size(); i++)
    {
        cout<<"Blob #"<<i<<" : "<<bn[i]<<endl;
    }
    return 0;
}
```

编译:

```
$ g++ -o netapp net_demo.cpp -I$CAFFE_ROOT/include -D CPU_ONLY -I
$CAFFE_ROOT/.build_release/src/ -L$CAFFE_ROOT/build/lib -lcaffe -lglog -lboost_system
-lprotobuf
```

运行:

```
$ ./netapp
WARNING: Logging before InitGoogleLogging() is written to STDERR
I0515 17:13:00.473947 45913 net.cpp:49] Initializing net from parameters:
name: "CaffeNet"
// .....大部分内容我们都见过了
I0515 17:13:00.591897 45913 net.cpp:274] Network initialization done.
#### Blob Names ####
Blob #0 : data
Blob #1 : conv1
Blob #2 : pool1
Blob #3 : norm1
Blob #4 : conv2
Blob #5 : pool2
Blob #6 : norm2
Blob #7 : conv3
Blob #8 : conv4
Blob #9 : conv5
Blob #10 : pool5
Blob #11 : fc6
Blob #12 : fc7
Blob #13 : fc8
Blob #14 : prob
```

```
#### Layer Names ####  
Layer #0 : data  
Layer #1 : conv1  
Layer #2 : relu1  
Layer #3 : pool1  
Layer #4 : norm1  
Layer #5 : conv2  
Layer #6 : relu2  
Layer #7 : pool2  
Layer #8 : norm2  
Layer #9 : conv3  
Layer #10 : relu3  
Layer #11 : conv4  
Layer #12 : relu4  
Layer #13 : conv5  
Layer #14 : relu5  
Layer #15 : pool5  
Layer #16 : fc6  
Layer #17 : relu6  
Layer #18 : drop6  
Layer #19 : fc7  
Layer #20 : relu7  
Layer #21 : drop7  
Layer #22 : fc8  
Layer #23 : prob
```

通过上面简单的例子，我们可以看到 Net 中既包括 Layer 对象，又包括 Blob 对象。其中 Blob 对象用于存放每个 Layer 输入/输出中间结果，Layer 则根据 Net 描述对指定的输入 Blob 进行某些计算处理（卷积、下采样、全连接、非线性变换、计算代价函数等），输出结果放到指定的输出 Blob 中。输入 Blob 和输出 Blob 可能为同一个。所有的 Layer 和 Blob 对象都用名字区分，同名的 Blob 表示同一个 Blob 对象，同名的 Layer 表示同一个 Layer 对象。而 Blob 和 Layer 同名则不代表它们有任何直接关系。

我们可以通过 `has_blob()`、`has_layer()` 函数来查询当前 Net 对象是否包含指定名字的 Blob 或 Layer 对象，如果返回值为真，则可以进一步调用 `blob_by_name()`、`layer_by_name()` 函数直接获取相应的 Blob 或 Layer 指针，进行一些操作（如提取某层计算输出特征或某个 Blob 中的权值）。

8.3.2 数据结构描述

依然先通过 `caffe.proto` 一窥“网络”的本质。

```
message NetParameter {
  optional string name = 1; // 网络名称
  repeated string input = 3; // 网络的输入 Blob 名称，可以有多个 Blob
  repeated BlobShape input_shape = 8; // 输入 Blob 的维度信息
  repeated int32 input_dim = 4; // 旧版的维度信息
  // 网络是否强制每个层执行后向传播计算。如果设置为 false，那么是否执行后向传播计算由网络结构、学习速率自动确定
  optional bool force_backward = 5 [default = false];
  // 网络的当前状态（包括 phase、level 以及 stage）
  optional NetState state = 6;
  // 在运行 Net::Forward、Net::Backward、Net::Update 时是否打印结果的调试信息
  optional bool debug_info = 7 [default = false];
  // 组成 Net 的所有层，每个层配置都包括连接属性与行为，由 LayerParameter 定义
  repeated LayerParameter layer = 100; // ID 设为 100，这样层描述会置于末尾
  // 已淘汰
  repeated V1LayerParameter layers = 2;
}
```

看似很短的 `proto` 描述，实际上对应的真实网络 `prototxt` 可以很长很长，关键在于可重复多次出现的 `LayerParameter layer` 这个字段。其他字段的功能基本都是辅助网络运行的，在代码中会看到更多的细节。

8.3.3 Net 是怎样绘成的

我们将 `Blob` 比作 Caffe 砖石，`Layer` 比作 Caffe 的墙面，那么 `Net` 更像是工匠手中的图纸，描述了每个墙面应当出现的位置，这样设计的房屋才足够牢固、抗震。为了达到这个目的，`Net` 实现时必然有一套用于记录 `Layer`、`Blob` 的数据结构。在表 8-1 中提前向大家公布一下这些数据结构的名字，免得看到时面生，错过与它们打交道的机会。

表 8-1 Net 中很有必要认识的几个名字

类 对 象	含 义
<code>layers_</code>	记录 Net <code>prototxt</code> 中出现的每个 <code>Layer</code>
<code>layer_names_</code>	记录 Net <code>prototxt</code> 中出现的每个 <code>Layer</code> 的名称
<code>layer_names_index_</code>	记录 Net <code>prototxt</code> 中每个 <code>Layer</code> 名称与顺序索引的对应关系

续表

类 对 象	含 义
layer_need_backward_	记录每个 Layer 是否需要反向传播过程
blobs_	记录 Net 中所有 Blob
blob_names_	记录每个 Blob 名称
blob_names_index_	记录每个 Blob 名称与顺序索引的对应关系
blob_need_backward_	记录每个 Blob 是否需要反向传播过程
bottom_vecs_	blobs_ 的影子，记录每个 Layer 的输入 Blob
bottom_id_vecs_	与 bottom_vecs_ 关联，用于在 blobs_ 中定位每个 Layer 的每个输入 Blob
bottom_need_backward_	与 bottom_vecs_ 关联，标志每个 Blob 是否需要反向传播过程
top_vecs_	blobs_ 的影子，记录每个 Layer 的输出 Blob
top_id_vecs_	与 top_vecs_ 关联，用于在 blobs_ 中定位每个 Layer 的每个输出 Blob
blob_loss_weights_	Net 中每个 Blob 对损失函数的投票因子。一般损失层为 1，其他层为 0
net_input_blob_indices_	Net 输入 Blob 在 blobs_ 中的索引
net_output_blob_indices_	Net 输出 Blob 在 blobs_ 中的索引
net_input_blobs_	Net 输入 Blob
net_output_blobs_	Net 输出 Blob
params_	Net 权值 Blob，用于存储网络权值
param_display_names_	Net 中权值 Blob 的名称
learnable_params_	Net 中可训练的权值 Blob
params_lr_	learnable_params_ 中每个元素的学习速率倍乘因子
has_params_lr_	标志 learnable_params_ 中每个元素是否有学习速率倍乘因子
params_weight_decay_	learnable_params_ 中每个元素的权值衰减倍乘因子
has_params_decay_	标志 learnable_params_ 中每个元素是否有权值衰减倍乘因子

看到上面有两类 Blob：以 param 开头的权值 Blob 和以 blob 开头的 Layer 输入/输出 Blob。它们虽然都是 Blob 类型，但在网络中的地位截然不同。权值 Blob 会随着学习过程而更新，归属于“模型”；Layer 输入/输出 Blob 则只会随网络输入变化，归属于“数据”。深度学习的目的就是不断从“数据”中获取知识，存储到“模型”中，应用于后来的“数据”。

Net 声明位于 include/caffe/net.hpp 中，内容如下：

```
template <typename Dtype>
class Net {
public:
```

```

// 显式构造函数
explicit Net(const NetParameter& param, const Net* root_net = NULL);
explicit Net(const string& param_file, Phase phase,
             const Net* root_net = NULL);
// 析构函数
virtual ~Net() {}

// 用 NetParameter 对象初始化 Net
void Init(const NetParameter& param);
// 运行前向传播, 输入 Blob 已经预先填充
const vector<Blob<Dtype>*>& ForwardPrefilled(Dtype* loss = NULL);
// Net 前向传播的几种形式
Dtype ForwardFromTo(int start, int end);
Dtype ForwardFrom(int start);
Dtype ForwardTo(int end);
const vector<Blob<Dtype>*>& Forward(const vector<Blob<Dtype>*>& bottom,
                                   Dtype* loss = NULL); // 指定输入 Blob 进行前向传播, 返回输出 Blob
string Forward(const string& input_blob_protos, Dtype* loss = NULL); // 指定序列化的输入
BlobProtoVector Forward(const string& input_blob_protos, Dtype* loss = NULL); // 指定序列化的输入
BlobProtoVector Forward(const string& input_blob_protos, Dtype* loss = NULL); // 指定序列化的输入
// 清零所有权值的 diff 域, 应在反向传播之前运行
void ClearParamDiffs();
// 几种不同形式的 Net 反向传播, 无须指定输入/输出 Blob, 因为在前向传播过程中已经建立连接
void Backward();
void BackwardFromTo(int start, int end);
void BackwardFrom(int start);
void BackwardTo(int end);
// 对 Net 中所有 Layer 自底向上进行变形, 无须运行一次前向传播就可以计算各层所需的 Blob 尺寸
void Reshape();
// 前向传播+反向传播, 输入为 Bottom Blob, 输出为 loss
Dtype ForwardBackward(const vector<Blob<Dtype>*>& bottom) {
    Dtype loss;
    Forward(bottom, &loss);
    Backward();
    return loss;
}
// 根据已经 (由 Solver) 准备好的 diff 值更新网络权值
void Update();

void ShareWeights();

```

```

// 从一个已训练好的 Net 获取共享权值
void ShareTrainedLayersWith(const Net* other);
void CopyTrainedLayersFrom(const NetParameter& param);
void CopyTrainedLayersFrom(const string trained_filename);
void CopyTrainedLayersFromBinaryProto(const string trained_filename);
void CopyTrainedLayersFromHDF5(const string trained_filename);
// 序列化一个 Net 到 ProtoBuffer
void ToProto(NetParameter* param, bool write_diff = false) const;
// 序列化一个 Net 到 HDF5
void ToHDF5(const string& filename, bool write_diff = false) const;

// 返回网络名称
inline const string& name() const { return name_; }
// 返回一组 Layer 名称
inline const vector<string>& layer_names() const { return layer_names_; }
// 返回一组 Blob 名称
inline const vector<string>& blob_names() const { return blob_names_; }
// 返回 blobs_
inline const vector<shared_ptr<Blob<Dtype>>>& blobs() const {
    return blobs_;
}
// 返回 layers_
inline const vector<shared_ptr<Layer<Dtype>>>& layers() const {
    return layers_;
}
// 返回当前阶段: TRAIN 或 TEST
inline Phase phase() const { return phase_; }
// 返回每个 Layer 的 Bottom Blob
inline const vector<vector<Blob<Dtype>*>>& bottom_vecs() const {
    return bottom_vecs_;
}
// 返回每个 Layer 的 Top Blob
inline const vector<vector<Blob<Dtype>*>>& top_vecs() const {
    return top_vecs_;
}
// 返回每个 Layer 的 Bottom Blob 是否需要反向传播过程
inline const vector<vector<bool>>& bottom_need_backward() const {
    return bottom_need_backward_;
}
}

```

```

// 返回每个 Blob 是否参与 loss 计算
inline const vector<Dtype>& blob_loss_weights() const {
    return blob_loss_weights_;
}

// 返回每个 Layer 是否需要反向传播计算
inline const vector<bool>& layer_need_backward() const {
    return layer_need_backward_;
}

// 返回所有权值
inline const vector<shared_ptr<Blob<Dtype>>>& params() const {
    return params_;
}

// 返回所有可训练权值
inline const vector<Blob<Dtype>*>& learnable_params() const {
    return learnable_params_;
}

// 返回可训练权值的学习速率倍乘因子
inline const vector<float>& params_lr() const { return params_lr_; }
inline const vector<bool>& has_params_lr() const { return has_params_lr_; }

// 返回可训练权值的衰减因子
inline const vector<float>& params_weight_decay() const {
    return params_weight_decay_;
}

inline const vector<bool>& has_params_decay() const {
    return has_params_decay_;
}

// 返回 Layer 名称与向量下标映射对
const map<string, int>& param_names_index() const {
    return param_names_index_;
}

// 返回权值所有者
inline const vector<int>& param_owners() const { return param_owners_; }

// 返回输入/输出 Blob 数目
inline int num_inputs() const { return net_input_blobs_.size(); }
inline int num_outputs() const { return net_output_blobs_.size(); }

// 返回输入 Blob
inline const vector<Blob<Dtype>*>& input_blobs() const {
    return net_input_blobs_;
}

```

```

// 返回输出 Blob
inline const vector<Blob<Dtype>*>& output_blobs() const {
    return net_output_blobs_;
}

// 返回输入 Blob 下标
inline const vector<int>& input_blob_indices() const {
    return net_input_blob_indices_;
}

// 返回输出 Blob 下标
inline const vector<int>& output_blob_indices() const {
    return net_output_blob_indices_;
}

// 查找当前网络是否包含某一名称 Blob
bool has_blob(const string& blob_name) const;
// 如果包含, 那么就请把它找出来
const shared_ptr<Blob<Dtype>> blob_by_name(const string& blob_name) const;
// 查找当前网络是否包含某一名称 Layer
bool has_layer(const string& layer_name) const;
// 如果包含, 那么就请把它找出来
const shared_ptr<Layer<Dtype>> layer_by_name(const string& layer_name) const;
// 设置 debug_info_
void set_debug_info(const bool value) { debug_info_ = value; }

// 下面这些函数是 Init 的好帮手
// 过滤掉用户指定的在某个阶段、级别、状态下不应包含的 Layer
static void FilterNet(const NetParameter& param,
    NetParameter* param_filtered);
// 判断网络状态是否满足网络规则
static bool StateMeetsRule(const NetState& state, const NetStateRule& rule,
    const string& layer_name);

protected:
// 为网络追加一个 Top Blob
void AppendTop(const NetParameter& param, const int layer_id,
    const int top_id, set<string>* available_blobs,
    map<string, int>* blob_name_to_idx);
// 为网络追加一个 Bottom Blob
int AppendBottom(const NetParameter& param, const int layer_id,
    const int bottom_id, set<string>* available_blobs,

```



```

        map<string, int>* blob_name_to_idx);
// 为网络追加一个权值 Blob
void AppendParam(const NetParameter& param, const int layer_id,
                 const int param_id);

// 显示输入调试信息
void InputDebugInfo(const int layer_id);
// 显示前向传播调试信息
void ForwardDebugInfo(const int layer_id);
// 显示反向传播调试信息
void BackwardDebugInfo(const int layer_id);
// 显示权值更新调试信息
void UpdateDebugInfo(const int param_id);

string name_; // 网络名称

Phase phase_; // 当前阶段 (TRAIN 或 TEST)

vector<shared_ptr<Layer<Dtype>>> layers_; // 网络中的独立层
vector<string> layer_names_; // 层名称
map<string, int> layer_names_index_; // 层名称与索引映射表
vector<bool> layer_need_backward_; // 标记某个层是否需要 BP

vector<shared_ptr<Blob<Dtype>>> blobs_; // 层与层中间传递数据的管道
vector<string> blob_names_; // Blob 名称
map<string, int> blob_names_index_; // Blob 名称与索引映射表
vector<bool> blob_need_backward_; // 标记某个 Blob 是否需要 BP
// bottom_vecs_ 存放每个层的输入 Blob, 实际上它并不是这些 Blob 的所有者 (所有者为 blobs_), 只是
// 存放了指针
vector<vector<Blob<Dtype>*>> bottom_vecs_;
vector<vector<int>> bottom_id_vecs_;
vector<vector<bool>> bottom_need_backward_;
// top_vecs_ 存放每个层的输出 Blob, 实际上它并不是这些 Blob 的所有者 (所有者为 blobs_), 只是存放了
// 指针
vector<vector<Blob<Dtype>*>> top_vecs_;
vector<vector<int>> top_id_vecs_;
// 每个 Blob 对全局损失函数 (目标函数) 的贡献权重
vector<Dtype> blob_loss_weights_;
vector<vector<int>> param_id_vecs_;

```

```

vector<int> param_owners_;
vector<string> param_display_names_;
vector<pair<int, int>> param_layer_indices_;
map<string, int> param_names_index_;
// 网络输入/输出 Blob 的索引
vector<int> net_input_blob_indices_;
vector<int> net_output_blob_indices_;
vector<Blob<Dtype>*> net_input_blobs_;
vector<Blob<Dtype>*> net_output_blobs_;
// 网络权值
vector<shared_ptr<Blob<Dtype>>> params_;
// 可训练的网络权值
vector<Blob<Dtype>*> learnable_params_;
// 从 params_ 到 learnable_params_ 的映射
// 当且仅当 params_[i] 为所有者时, learnable_param_ids_.size() == params_.size() 以及
learnable_params_[learnable_param_ids_[i]] == params_[i].get() 成立
// 否则, params_[i] 只是一个共享者, learnable_params_[learnable_param_ids_[i]] 给出了它的所
// 有者
vector<int> learnable_param_ids_;
// 学习速率倍增因子
vector<float> params_lr_;
vector<bool> has_params_lr_;
// 权值衰减因子
vector<float> params_weight_decay_;
vector<bool> has_params_decay_;
// 记录网络占用的内存大小
size_t memory_used_;
// 是否显示调试信息
bool debug_info_;
// 在数据并行条件下, 根网络是实际有效网络, 其他网络都间接引用了根网络
const Net* const root_net_;
DISABLE_COPY_AND_ASSIGN(Net); // 禁用拷贝构造函数、赋值运算函数
};

```

我们今天卖个关子, Net 具体实现代码暂不公布, 等到合适的机会再做深入分析。

8.4 机制和策略

代码读到这里, 读者若能建立如下观念, 则会更容易接受 Caffe 中 Net/Layer/Blob 这种分层

的设计模式。

在我们生活中普遍存在但又最容易被忽视的两个概念是：**机制和策略**。

一般来说，对于某客观事物，机制回答了“它能干啥”这个问题，策略则回答了“它怎么用”这个问题。

例 1：插座提供电源，是一种提供电能的机制，而电视机、洗衣机、电冰箱、空调等家用电器则提供使用电能的不同策略；

例 2：操作系统提供计算机的管理机制，而计算机用户提供如何管理计算机的策略；

例 3：在互联网应用中，服务器提供特定服务机制，客户端提供使用服务的策略；

例 4：超市提供品类齐全的商品机制，而消费者根据自身需求提供购买策略；

例 5：数学家提供逻辑严谨的理论机制，科学家提供使用理论指导实验的策略；

例 6：云计算厂商提供计算资源的机制，而生长在云上的大小应用厂商则提供使用计算资源的不同策略；

例 7：在工作中，你的主管掌握部门运作流程中绝大部分策略，例如哪个员工具有哪方面能力，适合干哪些工作，而你只是提供机制的螺丝钉，做好本职工作即可（年轻人好好搬砖，不要随便问大战略）。

回到 Caffe 源码上，我们发现 Blob 提供了数据容器的机制；而 Layer 则通过不同的策略使用该数据容器，实现多元化的计算处理过程，同时又提供了深度学习各种基本算法（卷积、下采样、损失函数计算等）的机制；Net 则利用 Layer 这些机制，组合为完整的深度学习模型，提供了更加丰富的学习策略。后面我们还会看到，Net 也是一种机制。

在阅读源码时，时刻记得目标是希望看到高层策略，还是底层机制？

8.5 练习题

1. Net 初始化时如何统计所需的存储空间？
2. 在 C++ 中如何禁用某个类的拷贝构造函数与赋值运算符重载？

答案尽在 Caffe 源码中。

3. 学习其他深度学习框架中对应的数据结构，了解其提供了哪些机制。

8.6 参考资料

[1] Siddhartha Rao. 21 天学通 C++. 2012

[2] <https://developers.google.com/protocol-buffers/docs/proto>

第 9 天

Caffe I/O 模块

今天我们学习 Caffe 的 I/O 模块，即与数据打交道的模块。

也许读者还记得，我们在运行 Caffe 例程前，首先需要将原始数据转换为 LMDB 格式，训练网络时则需要由数据读取层（DataLayer）不断地从 LMDB 读取数据，送入后续卷积、下采样等计算层。俗话说，“民以食为天”，Caffe I/O 模块的效率直接影响到处理效果。

9.1 数据读取层

Caffe 数据读取层（DataLayer）是 Layer 的派生类。除了读取 LMDB、LEVELDB 之外，也可以从原始图像直接读取（ImageDataLayer）。

9.1.1 数据结构描述

```
message DataParameter {
  // 输入数据使用的 DB 类型
  enum DB {
    LEVELDB = 0;  // 使用 LEVELDB
    LMDB = 1;     // 使用 LMDB
  }
  // 源数据的路径
  optional string source = 1;
  // 一个批量数据包含的图片数目
  optional uint32 batch_size = 4;
  // 随机跳过若干图片，跳跃数目为 rand_skip * rand(0, 1)
  optional uint32 rand_skip = 7 [default = 0];
```

```

// 默认输入数据使用 DB 类型，默认为 LEVELDB
optional DB backend = 8 [default = LEVELDB];
// scale、mean_file、crop_size、mirror 均为旧版参数，现已转移到 TransformationParameter
optional float scale = 2 [default = 1];
optional string mean_file = 3;
optional uint32 crop_size = 5 [default = 0];
optional bool mirror = 6 [default = false];
// 强制编码图像为三通道彩色图像
optional bool force_encoded_color = 9 [default = false];
// 预取队列（预先放到主机内存中的批量数，默认为 4 个 Batch）
optional uint32 prefetch = 10 [default = 4];
}

```

9.1.2 数据读取层实现

数据读取层声明位于 `include/caffe/data_layers.hpp` 中，如果需要单独使用该层，则应包含这个头文件。

```

// 基本数据层，派生于 Layer
template <typename Dtype>
class BaseDataLayer : public Layer<Dtype> {
public:
    // 显式构造函数
    explicit BaseDataLayer(const LayerParameter& param);
    // 层配置，实现通用层配置功能，之后调用 DataLayerSetUp 进行数据读取层的特别配置
    virtual void LayerSetUp(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    virtual void DataLayerSetUp(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top) {}
    // 数据读取层应被多个并行求解器共享
    virtual inline bool ShareInParallel() const { return true; }
    // 数据读取层没有 Bottom Blob，变形操作很简单
    virtual void Reshape(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top) {}
    // 反向传播函数不需要做任何操作
    virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom) {}
    virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom) {}
}

```

```

protected:
    // 数据预处理变换器参数
    TransformationParameter transform_param_;
    // 数据预处理变换器
    shared_ptr<DataTransformer<Dtype>> data_transformer_;
    // 是否输出标签数据
    bool output_labels_;
};

// 批量数据，用于存放数据读取层输出
template <typename Dtype>
class Batch {
public:
    // 包含两个 Blob: data_用于存放图片数据, label_用于存放标签
    Blob<Dtype> data_, label_;
};

// 带预取功能的数据读取层，派生于 BaseDataLayer 和 InternalThread
template <typename Dtype>
class BasePrefetchingDataLayer :
    public BaseDataLayer<Dtype>, public InternalThread {
public:
    // 显式构造函数
    explicit BasePrefetchingDataLayer(const LayerParameter& param);
    // 层设置函数
    void LayerSetUp(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    // 前向传播
    virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    // 反向传播
    virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);

    // 预取的数据批量数目
    static const int PREFETCH_COUNT = 3;
};

protected:
    virtual void InternalThreadEntry(); // 内部线程入口
    virtual void load_batch(Batch<Dtype>* batch) = 0; // 载入批量数据，纯虚函数

```

```

Batch<Dtype> prefetch_[PREFETCH_COUNT]; // 预取 Buffer
BlockingQueue<Batch<Dtype>*> prefetch_free_; // 空闲 Batch 队列
BlockingQueue<Batch<Dtype>*> prefetch_full_; // 已加载 Batch 队列

Blob<Dtype> transformed_data_; // 变换后的数据
};

```

数据读取层的实现位于 `src/caffe/layers/base_data_layer.cpp` 中，内容如下：

```

#include <boost/thread.hpp>
#include <string>
#include <vector>

#include "caffe/data_layers.hpp"
#include "caffe/net.hpp"
#include "caffe/util/io.hpp"

namespace caffe {
// 构造函数，初始化 Layer 参数、数据变换器参数
template <typename Dtype>
BaseDataLayer<Dtype>::BaseDataLayer(const LayerParameter& param)
    : Layer<Dtype>(param),
      transform_param_(param.transform_param()) {
}
// BaseDataLayer 层设置
template <typename Dtype>
void BaseDataLayer<Dtype>::LayerSetUp(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
    if (top.size() == 1) { // 判断输出 Blob 个数，若为 1 只输出 data，若为 2 则输出 data 和 label
        output_labels_ = false;
    } else {
        output_labels_ = true;
    }
    // 初始化数据变换器对象
    data_transformer_.reset(
        new DataTransformer<Dtype>(transform_param_, this->phase_));
    data_transformer_->InitRand(); // 生成随机数种子
    // 子类负责设置 Top Blob 形状
    DataLayerSetUp(bottom, top);
}

```



```

}

// BasePrefetchingDataLayer 构造函数
template <typename Dtype>
BasePrefetchingDataLayer<Dtype>::BasePrefetchingDataLayer(
    const LayerParameter& param)
    : BaseDataLayer<Dtype>(param),
      prefetch_free_(), prefetch_full_() {
    for (int i = 0; i < PREFETCH_COUNT; ++i) {
        prefetch_free_.push(&prefetch_[i]); // 将 Batch 对象都放入空闲队列中
    }
}

// BasePrefetchingDataLayer 层配置函数
template <typename Dtype>
void BasePrefetchingDataLayer<Dtype>::LayerSetUp(
    const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top) {
    BaseDataLayer<Dtype>::LayerSetUp(bottom, top);

    // 在开启数据预取线程前，通过调用 Blob 相应函数先进行 cudaMalloc，避免在多线程情况下同时进行
    // cudaMalloc，会导致 CUDA API 调用失败
    for (int i = 0; i < PREFETCH_COUNT; ++i) {
        prefetch_[i].data_.mutable_cpu_data();
        if (this->output_labels_) {
            prefetch_[i].label_.mutable_cpu_data();
        }
    }

    // 如果编译选项没有 CPU_ONLY，则需要编译 GPU 代码
    #ifndef CPU_ONLY
    if (Caffe::mode() == Caffe::GPU) {
        for (int i = 0; i < PREFETCH_COUNT; ++i) {
            prefetch_[i].data_.mutable_gpu_data();
            if (this->output_labels_) {
                prefetch_[i].label_.mutable_gpu_data(); // 功能同上
            }
        }
    }
    #endif

    DLOG(INFO) << "Initializing prefetch";
    this->data_transformer_->InitRand();
    StartInternalThread(); // 开启内部预取线程

```

```

    DLOG(INFO) << "Prefetch initialized.";
}
// 内部线程入口
template <typename Dtype>
void BasePrefetchingDataLayer<Dtype>::InternalThreadEntry() {
    // 创建 CUDA Stream, 非阻塞类型
#ifdef CPU_ONLY
    cudaStream_t stream;
    if (Caffe::mode() == Caffe::GPU) {
        CUDA_CHECK(cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking));
    }
#endif

    try {
        while (!must_stop()) { // 循环载入批量数据
            Batch<Dtype>* batch = prefetch_free_.pop(); // 拿到一个空闲 Batch
            load_batch(batch); // 载入批量数据
#ifdef CPU_ONLY
            if (Caffe::mode() == Caffe::GPU) {
                batch->data_.data().get()->async_gpu_push(stream);
                CUDA_CHECK(cudaStreamSynchronize(stream)); // 同步到 GPU
            }
#endif
            prefetch_full_.push(batch); // 加入到带负载的 Batch 队列中
        }
    } catch (boost::thread_interrupted&) { // 捕获到异常, 退出 while 循环
        // Interrupted exception is expected on shutdown
    }
#ifdef CPU_ONLY
    if (Caffe::mode() == Caffe::GPU) {
        CUDA_CHECK(cudaStreamDestroy(stream)); // 销毁 CUDA Stream
    }
#endif
}
// 前向传播函数
template <typename Dtype>
void BasePrefetchingDataLayer<Dtype>::Forward_cpu(
    const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top) {
    // 从带负载的 Batch 队列中取出一个 Batch 对象

```

```

Batch<Dtype>* batch = prefetch_full_.pop("Data layer prefetch queue empty");
// Top Blob 根据 Batch 形状进行变形
top[0]->Reshape(batch->data_.num(), batch->data_.channels(),
    batch->data_.height(), batch->data_.width());
// 将 Batch 中的数据拷贝到 Top Blob
caffe_copy(batch->data_.count(), batch->data_.cpu_data(),
    top[0]->mutable_cpu_data());
DLOG(INFO) << "Prefetch copied";
if (this->output_labels_) { // 如果需要输出标签数据
    // Top Blob 根据 Batch 中的 label_ 形状进行变形
    top[1]->ReshapeLike(batch->label_);
    // 将 Batch 中的数据拷贝到 Top Blob
    caffe_copy(batch->label_.count(), batch->label_.cpu_data(),
        top[1]->mutable_cpu_data());
}
// 将一个 Batch 数据送入 Net, 完成任务, 返回空闲队列接收下一批量数据
prefetch_free_.push(batch);
}

#ifdef CPU_ONLY
STUB_GPU_FORWARD(BasePrefetchingDataLayer, Forward);
#endif

INSTANTIATE_CLASS(BaseDataLayer);
INSTANTIATE_CLASS(BasePrefetchingDataLayer);

} // namespace caffe

```

9.2 数据变换器

Caffe 的数据变换器 (DataTransformer) 主要提供了对原始输入图像的预处理方法, 包括随机切块、随机镜像、幅度缩放、去均值、灰度/色度变换等。相信熟悉图像处理、OpenCV 的读者对上述操作并不陌生。

9.2.1 数据结构描述

```

message TransformationParameter {
    // 像素幅度缩放参数, 默认为 1, 即不缩放

```

```

optional float scale = 1 [default = 1];
// 图像随机镜像开关, 默认为 false, 即不进行镜像操作
optional bool mirror = 2 [default = false];
// 图像随机切块的大小, 默认为 0, 即不进行切块操作
optional uint32 crop_size = 3 [default = 0];
// 存储图像均值的文件
optional string mean_file = 4;
// 均值数值, 无须读取文件。若数目与图像通道数目相等, 则每个图像通道分别减去对应的均值; 如果只给出一个
// 值, 则每个图像通道都减去同一个均值
repeated float mean_value = 5;
// 强制为三通道彩色图像输入
optional bool force_color = 6 [default = false];
// 强制为单通道灰度图像输入
optional bool force_gray = 7 [default = false];
}

```

9.2.2 数据变换器的实现

数据变换器声明头文件位于 `include/caffe/data_transformer.hpp` 中, 如果需要单独使用该模块, 应包含这个头文件。文件内容如下:

```

#ifndef CAFFE_DATA_TRANSFORMER_HPP
#define CAFFE_DATA_TRANSFORMER_HPP

#include <vector>

#include "caffe/blob.hpp"
#include "caffe/common.hpp"
#include "caffe/proto/caffe.pb.h"

namespace caffe {
// DataTransformer 类声明
template <typename Dtype>
class DataTransformer {
public:
// 显式构造函数
explicit DataTransformer(const TransformationParameter& param, Phase phase);
// 析构函数
virtual ~DataTransformer() {}
// 初始化随机数种子函数

```

```

void InitRand();

// 将数据读取层中 transform_param 块所声明的变换应用到输入数据中
// 函数重载，以适应多种输入数据源
void Transform(const Datum& datum, Blob<Dtype>* transformed_blob);
void Transform(const vector<Datum>& datum_vector,
               Blob<Dtype>* transformed_blob);
void Transform(const vector<cv::Mat>& mat_vector,
               Blob<Dtype>* transformed_blob);
void Transform(const cv::Mat& cv_img, Blob<Dtype>* transformed_blob);
void Transform(Blob<Dtype>* input_blob, Blob<Dtype>* transformed_blob);
void Transform(const Datum& datum, Dtype* transformed_data);
// 获取执行 Transform 后的输出 Blob 形状
vector<int> InferBlobShape(const Datum& datum);
vector<int> InferBlobShape(const vector<Datum>& datum_vector);
vector<int> InferBlobShape(const vector<cv::Mat>& mat_vector);
vector<int> InferBlobShape(const cv::Mat& cv_img);

protected:
// 产生取值{0, 1, ..., n-1}的随机整数，服从均匀分布
virtual int Rand(int n);
// 变换参数，该数据结构由 ProtoBuffer 工具自动生成
TransformationParameter param_;
// 随机数生成器，声明在 include/caffe/common.hpp 中
shared_ptr<Caffe::RNG> rng_;
// 当前运行阶段，可能为 TRAIN 或 TEST。阶段不同，执行变换会有差异
Phase phase_;
// 均值图像，用于从均值文件中读取
Blob<Dtype> data_mean_;
// 均值数值，用于从 param_ 中提取
vector<Dtype> mean_values_;
};

} // namespace caffe

#endif // CAFFE_DATA_TRANSFORMER_HPP_

```

数据变换器的实现文件位于 src/caffe/data_transformer.cpp，我们来深入阅读一下。

```
#include <opencv2/core/core.hpp>
```

```

#include <string>
#include <vector>

#include "caffe/data_transformer.hpp"
#include "caffe/util/io.hpp"
#include "caffe/util/math_functions.hpp"
#include "caffe/util/rng.hpp"

namespace caffe {
// 构造函数
template<typename Dtype>
DataTransformer<Dtype>::DataTransformer(const TransformationParameter& param,
    Phase phase)
    : param_(param), phase_(phase) {    // 初始化 param_ 和 phase_
// 查看是否使用均值文件
if (param_.has_mean_file()) {
    // 如果 param_ 中指定了均值文件，又指定了均值数值，则报错，只能二选一
    CHECK_EQ(param_.mean_value_size(), 0) <<
        "Cannot specify mean_file and mean_value at the same time";
    const string& mean_file = param.mean_file(); // 获取均值文件名
    .....
    // 从均值文件中读取数据到 blob_proto 对象中
    BlobProto blob_proto;
    ReadProtoFromBinaryFileOrDie(mean_file.c_str(), &blob_proto);
    // 从 blob_proto 将均值反序列化到 data_mean_ 内存中
    data_mean_.FromProto(blob_proto);
}
// 查看是否使用均值数值
if (param_.mean_value_size() > 0) {
    CHECK(param_.has_mean_file() == false) <<
        "Cannot specify mean_file and mean_value at the same time";
    for (int c = 0; c < param_.mean_value_size(); ++c) {
        mean_values_.push_back(param_.mean_value(c)); // 从 param_ 中读取均值数值，不再读取均值文件
    }
}
}

// 变换函数，从众多重载函数中，我们选择一个重点讲解，其他的计算流程都类似
// 下面函数使用了 Datum 作为输入，这个结构体我们可以从 caffe.proto 中一窥究竟

```

```

/*
// Datum 用来从 LMDB/LEVELDB 中读取数据，或将数据写入 LMDB/LEVELDB，和 BlobProto 有相似的功能，只是 BlobProto 用于模型权重序列化/反序列化，而 Datum 专为数据或特征图 (feature map) 提供序列化/反序列化服务

message Datum {
    // 数据维度信息, channels * height * width
    optional int32 channels = 1;
    optional int32 height = 2;
    optional int32 width = 3;
    // 图像数据，以字节类型存储
    optional bytes data = 4;
    // 标签数据，统一用 int32 类型存储
    optional int32 label = 5;
    // 可选，图像数据也可以用 float 类型存储
    repeated float float_data = 6;
    // 是否为编码数据，默认不是
    optional bool encoded = 7 [default = false];
}

*/

// 下面函数输入为 Datum，输出为数据指针
template<typename Dtype>
void DataTransformer<Dtype>::Transform(const Datum& datum,
Dtype* transformed_data) {
    // 获得 datum 数据字符串、维度信息
    const string& data = datum.data();
    const int datum_channels = datum.channels();
    const int datum_height = datum.height();
    const int datum_width = datum.width();
    // 从 param_ 获取处理参数，如切块大小、幅度缩放、随机镜像、图像均值等
    const int crop_size = param_.crop_size();
    const Dtype scale = param_.scale();
    const bool do_mirror = param_.mirror() && Rand(2);
    const bool has_mean_file = param_.has_mean_file();
    const bool has_uint8 = data.size() > 0;
    const bool has_mean_values = mean_values_.size() > 0;

    CHECK_GT(datum_channels, 0); // 保证输入数据通道数大于 0
    CHECK_GE(datum_height, crop_size); // 保证输入数据宽和高大于切块大小
    CHECK_GE(datum_width, crop_size);
    // 获得图像均值

```

```

Dtype* mean = NULL;
if (has_mean_file) { // 若指定了图像均值文件
    // 保证图像均值的维度与输入图像数据的维度完全相同
    CHECK_EQ(datum_channels, data_mean_.channels());
    CHECK_EQ(datum_height, data_mean_.height());
    CHECK_EQ(datum_width, data_mean_.width());
    mean = data_mean_.mutable_cpu_data(); // 夺取图像均值数据控制权
}

if (has_mean_values) { // 若没有指定图像均值文件，而是直接给出数值
    // 保证均值数值维度为 1，或与输入图像数据的 channels 数目相同
    CHECK(mean_values_.size() == 1 || mean_values_.size() == datum_channels) <<
        "Specify either 1 mean_value or as many as channels: " << datum_channels;
    if (datum_channels > 1 && mean_values_.size() == 1) {
        // 若均值数值维度为 1，而输入数据 channels 数目大于 1，则重复该值 channels 次
        for (int c = 1; c < datum_channels; ++c) {
            mean_values_.push_back(mean_values_[0]);
        }
    }
}

// 输入图像宽和高
int height = datum_height;
int width = datum_width;
// 开始图像切块
int h_off = 0;
int w_off = 0;
if (crop_size) { // crop_size 不为 0，则进行切块；若为 0 表示不切块
    height = crop_size;
    width = crop_size;
    // 训练阶段随机切块
    if (phase_ == TRAIN) {
        h_off = Rand(datum_height - crop_size + 1); // 切块的 height 偏移量
        w_off = Rand(datum_width - crop_size + 1); // 切块的 width 偏移量
    } else { // 测试阶段只切取图像中心位置
        h_off = (datum_height - crop_size) / 2;
        w_off = (datum_width - crop_size) / 2;
    }
}

Dtype datum_element; // 存放输入图像的像素值

```



```

int top_index, data_index; // 分别存放输出 index、输入 index
for (int c = 0; c < datum_channels; ++c) {
    for (int h = 0; h < height; ++h) {
        for (int w = 0; w < width; ++w) {
            data_index = (c * datum_height + h_off + h) * datum_width + w_off + w;
            if (do_mirror) { // 若需要镜像操作，则对输出 index 设置 width 反向
                top_index = (c * height + h) * width + (width - 1 - w);
            } else {
                top_index = (c * height + h) * width + w;
            }
            if (has_uint8) { // 若 datum 中使用 uint8 存储图像数据，需要转换为 float
                datum_element =
                    static_cast<Dtype>(static_cast<uint8_t>(data[data_index]));
            } else {
                datum_element = datum.float_data(data_index);
            }
            if (has_mean_file) { // 若指定了均值文件
                transformed_data[top_index] =
                    (datum_element - mean[data_index]) * scale; // 执行去均值、幅度缩放
            } else {
                if (has_mean_values) { // 若指定了均值数值
                    transformed_data[top_index] =
                        (datum_element - mean_values[c]) * scale; // 去均值、幅度缩放
                } else {
                    transformed_data[top_index] = datum_element * scale; // 不去均值，只做幅度缩放
                }
            }
        }
    }
}

// 与上面函数类似，只是输出变为 Blob
template<typename Dtype>
void DataTransformer<Dtype>::Transform(const Datum& datum,
                                       Blob<Dtype>* transformed_blob) {
    // 如果 datum 是经过编码的图像，则先解码
    if (datum.encoded()) {
        CHECK(!(param_.force_color() && param_.force_gray()))
        << "cannot set both force_color and force_gray";
    }
}

```

```

cv::Mat cv_img;
if (param_.force_color() || param_.force_gray()) {

    cv_img = DecodeDatumToCVMat(datum, param_.force_color());
} else {
    cv_img = DecodeDatumToCVMatNative(datum);
}
// 将 cv::image 变换为 Blob
return Transform(cv_img, transformed_blob);
} else {
    if (param_.force_color() || param_.force_gray()) {
        LOG(ERROR) << "force_color and force_gray only for encoded datum";
    }
}

const int crop_size = param_.crop_size();
const int datum_channels = datum.channels();
const int datum_height = datum.height();
const int datum_width = datum.width();

// 检查维度
const int channels = transformed_blob->channels();
const int height = transformed_blob->height();
const int width = transformed_blob->width();
const int num = transformed_blob->num();

CHECK_EQ(channels, datum_channels);
CHECK_LE(height, datum_height);
CHECK_LE(width, datum_width);
CHECK_GE(num, 1);

if (crop_size) {
    CHECK_EQ(crop_size, height);
    CHECK_EQ(crop_size, width);
} else {
    CHECK_EQ(datum_height, height);
    CHECK_EQ(datum_width, width);
}

```

```

Dtype* transformed_data = transformed_blob->mutable_cpu_data();
Transform(datum, transformed_data); // 参数变换完毕, 调用现有函数
}

// 对一组 datum 进行变换
template<typename Dtype>
void DataTransformer<Dtype>::Transform(const vector<Datum>& datum_vector,
                                       Blob<Dtype>* transformed_blob) {
    const int datum_num = datum_vector.size();
    const int num = transformed_blob->num();
    const int channels = transformed_blob->channels();
    const int height = transformed_blob->height();
    const int width = transformed_blob->width();

    CHECK_GT(datum_num, 0) << "There is no datum to add";
    CHECK_LE(datum_num, num) <<
        "The size of datum_vector must be no greater than transformed_blob->num()";
    Blob<Dtype> uni_blob(1, channels, height, width); // 临时 Blob
    // 依次对每个 datum 进行变换, 放入对应的 Blob 中
    for (int item_id = 0; item_id < datum_num; ++item_id) {
        int offset = transformed_blob->offset(item_id);
        uni_blob.set_cpu_data(transformed_blob->mutable_cpu_data() + offset);
        Transform(datum_vector[item_id], &uni_blob);
    }
}

// 对一组输入 cv::Mat 对象进行变换, 放入 Blob 中
template<typename Dtype>
void DataTransformer<Dtype>::Transform(const vector<cv::Mat>& mat_vector,
                                       Blob<Dtype>* transformed_blob) {
    const int mat_num = mat_vector.size();
    const int num = transformed_blob->num();
    const int channels = transformed_blob->channels();
    const int height = transformed_blob->height();
    const int width = transformed_blob->width();

    CHECK_GT(mat_num, 0) << "There is no MAT to add";
    CHECK_EQ(mat_num, num) <<
        "The size of mat_vector must be equals to transformed_blob->num()";
    Blob<Dtype> uni_blob(1, channels, height, width);
    for (int item_id = 0; item_id < mat_num; ++item_id) {

```

```

    int offset = transformed_blob->offset(item_id);
    uni_blob.set_cpu_data(transformed_blob->mutable_cpu_data() + offset);
    Transform(mat_vector[item_id], &uni_blob);
}
}

// 对一个 cv::Mat 对象进行变换
template<typename Dtype>
void DataTransformer<Dtype>::Transform(const cv::Mat& cv_img,
                                       Blob<Dtype>* transformed_blob) {
    const int crop_size = param_.crop_size();
    const int img_channels = cv_img.channels();
    const int img_height = cv_img.rows;
    const int img_width = cv_img.cols;

    // 检查维度
    const int channels = transformed_blob->channels();
    const int height = transformed_blob->height();
    const int width = transformed_blob->width();
    const int num = transformed_blob->num();

    CHECK_EQ(channels, img_channels);
    CHECK_LE(height, img_height);
    CHECK_LE(width, img_width);
    CHECK_GE(num, 1);

    CHECK(cv_img.depth() == CV_8U) << "Image data type must be unsigned byte";

    const Dtype scale = param_.scale();
    const bool do_mirror = param_.mirror() && Rand(2);
    const bool has_mean_file = param_.has_mean_file();
    const bool has_mean_values = mean_values_.size() > 0;

    CHECK_GT(img_channels, 0);
    CHECK_GE(img_height, crop_size);
    CHECK_GE(img_width, crop_size);

    Dtype* mean = NULL;
    if (has_mean_file) {
        CHECK_EQ(img_channels, data_mean_.channels());

```

```

CHECK_EQ(img_height, data_mean_.height());
CHECK_EQ(img_width, data_mean_.width());
mean = data_mean_.mutable_cpu_data();
}

if (has_mean_values) {
    CHECK(mean_values_.size() == 1 || mean_values_.size() == img_channels) <<
        "Specify either 1 mean_value or as many as channels: " << img_channels;
    if (img_channels > 1 && mean_values_.size() == 1) {
        // 复制均值数值, 便于操作
        for (int c = 1; c < img_channels; ++c) {
            mean_values_.push_back(mean_values_[0]);
        }
    }
}

int h_off = 0;
int w_off = 0;
// 用 OpenCV 实现图像切块
cv::Mat cv_cropped_img = cv_img;
if (crop_size) {
    CHECK_EQ(crop_size, height);
    CHECK_EQ(crop_size, width);
    // 只有训练阶段才会做随机切块
    if (phase_ == TRAIN) {
        h_off = Rand(img_height - crop_size + 1);
        w_off = Rand(img_width - crop_size + 1);
    } else {
        h_off = (img_height - crop_size) / 2;
        w_off = (img_width - crop_size) / 2;
    }
    cv::Rect roi(w_off, h_off, crop_size, crop_size);
    cv_cropped_img = cv_img(roi);
} else {
    CHECK_EQ(img_height, height);
    CHECK_EQ(img_width, width);
}

CHECK(cv_cropped_img.data);

```

```

Dtype* transformed_data = transformed_blob->mutable_cpu_data();
int top_index;
for (int h = 0; h < height; ++h) {
    const uchar* ptr = cv_cropped_img.ptr<uchar>(h);
    int img_index = 0;
    for (int w = 0; w < width; ++w) {
        for (int c = 0; c < img_channels; ++c) {
            if (do_mirror) {
                top_index = (c * height + h) * width + (width - 1 - w);
            } else {
                top_index = (c * height + h) * width + w;
            }
            // int top_index = (c * height + h) * width + w;
            Dtype pixel = static_cast<Dtype>(ptr[img_index++]);
            if (has_mean_file) {
                int mean_index = (c * img_height + h_off + h) * img_width + w_off + w;
                transformed_data[top_index] =
                    (pixel - mean[mean_index]) * scale;
            } else {
                if (has_mean_values) {
                    transformed_data[top_index] =
                        (pixel - mean_values[c]) * scale;
                } else {
                    transformed_data[top_index] = pixel * scale;
                }
            }
        }
    }
}

// 输入是Blob, 输出也是Blob
template<typename Dtype>
void DataTransformer<Dtype>::Transform(Blob<Dtype>* input_blob,
                                       Blob<Dtype>* transformed_blob) {
    const int crop_size = param_.crop_size();
    const int input_num = input_blob->num();
    const int input_channels = input_blob->channels();
    const int input_height = input_blob->height();
    const int input_width = input_blob->width();

```

```

if (transformed_blob->count() == 0) {
    // 初始化变换后的 Blob 形状
    if (crop_size) {
        transformed_blob->Reshape(input_num, input_channels,
                                   crop_size, crop_size);
    } else {
        transformed_blob->Reshape(input_num, input_channels,
                                   input_height, input_width);
    }
}

const int num = transformed_blob->num();
const int channels = transformed_blob->channels();
const int height = transformed_blob->height();
const int width = transformed_blob->width();
const int size = transformed_blob->count();

CHECK_LE(input_num, num);
CHECK_EQ(input_channels, channels);
CHECK_GE(input_height, height);
CHECK_GE(input_width, width);

const Dtype scale = param_.scale();
const bool do_mirror = param_.mirror() && Rand(2);
const bool has_mean_file = param_.has_mean_file();
const bool has_mean_values = mean_values_.size() > 0;

int h_off = 0;
int w_off = 0;
if (crop_size) {
    CHECK_EQ(crop_size, height);
    CHECK_EQ(crop_size, width);
    // 只有训练阶段才会做随机切块
    if (phase_ == TRAIN) {
        h_off = Rand(input_height - crop_size + 1);
        w_off = Rand(input_width - crop_size + 1);
    }
}

```

```

    } else {
        h_off = (input_height - crop_size) / 2;
        w_off = (input_width - crop_size) / 2;
    }
} else {
    CHECK_EQ(input_height, height);
    CHECK_EQ(input_width, width);
}

Dtype* input_data = input_blob->mutable_cpu_data();
if (has_mean_file) {
    CHECK_EQ(input_channels, data_mean_.channels());
    CHECK_EQ(input_height, data_mean_.height());
    CHECK_EQ(input_width, data_mean_.width());
    for (int n = 0; n < input_num; ++n) {
        int offset = input_blob->offset(n);
        caffe_sub(data_mean_.count(), input_data + offset,
                  data_mean_.cpu_data(), input_data + offset);
    }
}

if (has_mean_values) {
    CHECK(mean_values_.size() == 1 || mean_values_.size() == input_channels) <<
        "Specify either 1 mean_value or as many as channels: " << input_channels;
    if (mean_values_.size() == 1) {
        caffe_add_scalar(input_blob->count(), -(mean_values_[0]), input_data);
    } else {
        for (int n = 0; n < input_num; ++n) {
            for (int c = 0; c < input_channels; ++c) {
                int offset = input_blob->offset(n, c);
                caffe_add_scalar(input_height * input_width, -(mean_values_[c]),
                                input_data + offset);
            }
        }
    }
}

Dtype* transformed_data = transformed_blob->mutable_cpu_data();

```



```

for (int n = 0; n < input_num; ++n) {
    int top_index_n = n * channels;
    int data_index_n = n * channels;
    for (int c = 0; c < channels; ++c) {
        int top_index_c = (top_index_n + c) * height;
        int data_index_c = (data_index_n + c) * input_height + h_off;
        for (int h = 0; h < height; ++h) {
            int top_index_h = (top_index_c + h) * width;
            int data_index_h = (data_index_c + h) * input_width + w_off;
            if (do_mirror) {
                int top_index_w = top_index_h + width - 1;
                for (int w = 0; w < width; ++w) {
                    transformed_data[top_index_w-w] = input_data[data_index_h + w];
                }
            } else {
                for (int w = 0; w < width; ++w) {
                    transformed_data[top_index_h + w] = input_data[data_index_h + w];
                }
            }
        }
    }
}

if (scale != Dtype(1)) {
    DLOG(INFO) << "Scale: " << scale;
    caffe_scal(size, scale, transformed_data);
}
}

```

// 获得数据变换输出 Blob 尺寸

```

template<typename Dtype>
vector<int> DataTransformer<Dtype>::InferBlobShape(const cv::Mat& cv_img) {
    const int crop_size = param_.crop_size();
    const int img_channels = cv_img.channels();
    const int img_height = cv_img.rows;
    const int img_width = cv_img.cols;
    // 检查维度
    CHECK_GT(img_channels, 0);
}

```

```

CHECK_GE(img_height, crop_size);
CHECK_GE(img_width, crop_size);
// 创建 BlobShape 对象
vector<int> shape(4);
shape[0] = 1;
shape[1] = img_channels;
shape[2] = (crop_size)? crop_size: img_height;
shape[3] = (crop_size)? crop_size: img_width;
return shape;
}

// 初始化随机数种子
template <typename Dtype>
void DataTransformer<Dtype>::InitRand() {
    // 如果在初始化参数中要求对输入进行随机镜像操作，或者在训练阶段需要随机切块，
    // 那么需要初始化随机数种子
    const bool needs_rand = param_.mirror() ||
        (phase_ == TRAIN && param_.crop_size());
    if (needs_rand) {
        const unsigned int rng_seed = caffe_rng_rand();
        rng_.reset(new Caffe::RNG(rng_seed));
    } else {
        rng_.reset();
    }
}

// 生成 0 ~ n-1 之间的随机数
template <typename Dtype>
int DataTransformer<Dtype>::Rand(int n) {
    CHECK(rng_);
    CHECK_GT(n, 0);
    caffe::rng_t* rng =
        static_cast<caffe::rng_t*>(rng_>generator());
    return ((*rng)() % n);
}

INSTANTIATE_CLASS(DataTransformer);

} // namespace caffe

```

9.3 练习题

1. 阅读剩下的 `memory_data_layer.cpp`、`window_data_layer.cpp`。
2. 试着实现 `hdfs_data_layer`。
3. 试着用 Matlab 实现数据变换。

第 10 天

Caffe 模型

我们在上篇就讲过，一个完整的深度学习系统最核心的两个方面是**数据**和**模型**。今天我们主要关注模型。一个深度学习模型通常由三部分参数组成：

- 可学习参数 (Learnable Parameter)，又称可训练参数、神经网络权系数、权重，其数值由模型初始化参数、误差反向传播过程控制，一般不可人工干预。
- 结构参数 (Archetecture Parameter)，包括卷积层/全连接层/下采样层数目、卷积核数目、卷积核大小等描述网络结构的参数，一旦设定好，在网络训练阶段不能更改；值得注意的是，训练阶段网络结构参数和预测阶段结构参数很可能不同。
- 训练超参数 (Hyper-Parameter)，用来控制网络训练收敛的参数，训练阶段可以自动或手动调节以获得更好的效果，预测阶段不需要该参数。

在 Caffe 中，一个模型的三部分参数分别由不同模块定义和实现：

- 可学习参数在内存中使用 Blob 对象保持，必要时以二进制 ProtoBuffer 文件 (*.caffemodel) 形态序列化并存储于磁盘上，便于进一步微调 (finetune，又称精调)、共享 (例如参数服务器 Parameter Server, PS)、性能评估 (benchmark)。
- 结构参数使用 ProtoBuffer 文本格式 (*.prototxt) 描述，网络初始化时通过该描述文件构建 Net 对象、Layer 对象形成有向无环图结构，在 Layer 与 Layer 之间、Net 输入源和输出阱均为持有数据和中间结果的 Blob 对象。
- 训练超参数同样使用 ProtoBuffer 文本格式 (*.prototxt) 描述，训练阶段利用该描述文件构建求解器 (Solver) 对象，该对象按照一定规则在训练网络时自动调节这些超参数值。

下面我们一一展开进行介绍。

今天我们将在第 6 天介绍的 MNIST 例程中对 LeNet-5 模型稍加修改，变成如图 10-1 所示的逻辑回归（Logistic Regression，LR）分类器。

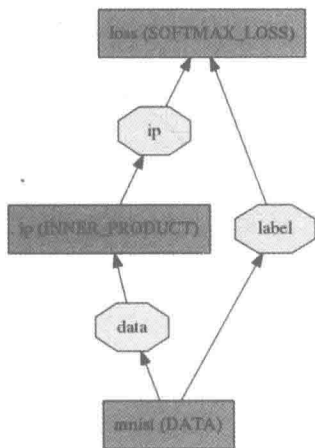


图10-1 LR分类器用于MNIST

10.1 prototxt 表示

复制一份 examples/mnist/lenet_train_test.prototxt，重命名为 lenet_lr.prototxt，修改内容如下：

```

name: "LeNet"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    batch_size: 64
  }
}

```

```
    backend: LMDB
  }
}
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
layer {
  name: "ip"
  type: "InnerProduct"
  bottom: "data"
  top: "ip"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

```

    }
}
layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
}

```

复制一份 `examples/mnist/lenet_solver.prototxt`，重命名为 `lenet_lr_solver.prototxt`，修改内容如下：

```

net: "examples/mnist/lenet_lr.prototxt"
test_iter: 100
test_interval: 500
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
lr_policy: "inv"
gamma: 0.0001
power: 0.75
display: 100
max_iter: 10000
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
solver_mode: CPU

```

运行训练，在命令行输入：

```
$ ./build/tools/caffe train --solver=examples/mnist/lenet_lr_solver.prototxt
```

经过训练，可以获得在测试集上分类准确率为 92.28% 的模型。相比 LeNet-5 而言准确率降低了，这也符合直觉，因为将模型简化后参数变少，层数变少，网络表达能力变差。我们今天不关注准确率，只关注模型的表达方式。

10.2 内存中的表示

从运行 log 文件可以追踪模型是如何从 prototxt 描述变为内存中表示方式的。

看到这行：

```
I0410 18:28:58.323169 52982 solver.cpp:91] Creating training net from net file:
examples/mnist/lenet_lr.prototxt
// . . . 不要在意这些细节
I0410 18:28:58.323587 52982 net.cpp:49] Initializing net from parameters:
```

追踪 solver.cpp 的第 91 行，看到如下代码：

```
// 在 solver.hpp 中声明了 SolverParameterparam_
// 它是 ProtoBuffer 工具生成的结构体，用来解析 lenet_lr_solver.prototxt
// 可参考 5.1 节例程，熟悉该结构体的用法
NetParameter net_param;
// . . .
if (param_.has_net()) { // 如果 lenet_lr_solver.prototxt 中指定了网络描述 prototxt 文件，这里为 true
    LOG_IF(INFO, Caffe::root_solver()) // 打印 log
        << "Creating training net from net file: "<< param_.net();
    // 这里 param_.net() 会返回 examples/mnist/lenet_lr.prototxt
    ReadNetParamsFromFileOrDie(param_.net(), &net_param);
    // 读取并解析 lenet_lr.prototxt 内容，将网络参数载入 net_param 结构体
    // 该函数是 ProtoBuffer 工具完成文本到结构体变量转换的
}
```

读者可以继续跟踪 net_param 的去向。

10.3 磁盘上的表示

Caffe 使用 ProtoBuffer 二进制文件有最小文件尺寸，并由 ProtoBuffer 工具自动生成高效的序列化/反序列化接口（多语言支持，包括 C++、Java、Python），以及可读性好、兼容二进制文

件的文本格式文件。

我们仍然从运行 log 查找线索：

```
I0410 18:29:05.610750 52982 solver.cpp:454] Snapshotting to binary proto file
examples/mnist/lenet_iter_10000.caffemodel
I0410 18:29:05.615471 52982 sgd_solver.cpp:273] Snapshotting solver state to binary proto
file examples/mnist/lenet_iter_10000.solverstate
```

其中，`.caffemodel` 文件是在特定训练间隙保存的二进制文件，包含当前网络各层的权值状态；而 `.solverstate` 是与 `.caffemodel` 一起产生的二进制文件，包含从上次停止点恢复训练模型所需的信息。我们具体看下列代码。

追踪 `solver.cpp` 的第 454 行，上下文信息如下：

```
// 在 solver.hpp 中声明了 SolverParameterparam_
// 它是 ProtoBuffer 工具生成的结构体，用来解析 lenet_lr_solver.prototxt
// 可参考 5.1 节例程，熟悉该结构体的用法
template<typename Dtype>
string Solver<Dtype>::SnapshotToBinaryProto() {
    string model_filename = SnapshotFilename(".caffemodel"); // 得到模型文件名
    LOG(INFO) << "Snapshotting to binary proto file " << model_filename;
    NetParameter net_param;
    net_ -> ToProto(&net_param, param_.snapshot_diff());
    // 将 net_ 转换为 NetParameter
    WriteProtoToBinaryFile(net_param, model_filename);
    // 写入 ProtoBuffer 二进制文件，这里是 lenet_iter_10000.caffemodel
    return model_filename;
}
```

追踪 `sgd_solver.cpp` 的第 273 行，上下文信息如下：

```
template<typename Dtype>
void SGDSolver<Dtype>::SnapshotSolverStateToBinaryProto(
    const string& model_filename) {
    SolverState state; // 创建一个序列化对象
    state.set_iter(this->iter_); // 记录当前迭代次数
    state.set_learned_net(model_filename); // 记录网络描述文件
    state.set_current_step(this->current_step_); // 记录当前步进值
    state.clear_history(); // 清空容器，准备接纳新内容
    for (int i = 0; i < history_.size(); ++i) {
        // 记录权值的历史信息
```

```

    BlobProto* history_blob = state.add_history();
    history_[i]->ToProto(history_blob);
}
stringsnapshot_filename = Solver<Dtype>::SnapshotFilename(".solverstate");
LOG(INFO)
    << "Snapshotting solver state to binary proto file "<<snapshot_filename;
WriteProtoToBinaryFile(state, snapshot_filename.c_str());
// 将 SolverState 对象写入二进制文件 (*.solverstate)
}

```

从磁盘上将模型、求解器状态文件载入内存的过程与上面代码刚好相反，读者可自行跟踪阅读。

10.4 Caffe Model Zoo

对于前面我们运行的简单模型，可以从头训练（from scratch）。然而，对于规模更大、结构更复杂的模型，从头训练需要解决两个问题：首先是硬件计算能力。模型训练十分消耗计算资源，使用普通计算机需要相当长的时间，不经济；而且世界上每个研究机构都从头训练，重复性工作太多，不环保。其次是调参能力。同样的模型设计，可能每个人训练结果都不一致，中间调参是项技术活，控制不当会引起训练发散或训练不充分，无法达到理想的分类效果。

为了解决上述问题，Caffe Model Zoo 则提供了一个分享模型的平台，世界各地的研究人员都可以把自己的训练成果共享给社区中更多的人使用，节省人力、物力。

今天我们也站在前人的肩膀上，运行一个基于已训练模型的图片分类例程。我们首先需要下载几个文件。

下载 meta 数据到当前目录：

```

$ cd data/ilsvrc12/
$ ./get_ilsvrc_aux.sh

```

下载 caffe 模型：

```

$ cd ../../models/bvlc_reference_caffenet/
$ wget http://dl.caffe.berkeleyvision.org/bvlc_reference_caffenet.caffemodel

```

回到根目录执行：

```

$ ./build/examples/cpp_classification/classification.bin\

```

```
models/bvlc_reference_caffenet/deploy.prototxt\
models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel \
data/ilsvrc12/imagenet_mean.binaryproto\
data/ilsvrc12/synset_words.txt \
examples/images/cat.jpg
```

命令行解释如下：

```
$ ./build/examples/cpp_classification/classification.bin
Usage: ./build/examples/cpp_classification/classification.bin \ // 二进制程序名
deploy.prototxt          // 模型描述文件
network.caffemodel       // *.caffemodel 模型权值文件
mean.binaryproto         // 图像均值文件
labels.txt               // 图像类别标签信息
img.jpg                  // 输入待分类图像
```

打开输入图像 examples/images/cat.jpg，如图 10-2 所示。



图10-2 cat.jpg

命令行输出的预测结果为：

```
----- Prediction for examples/images/cat.jpg -----
0.3132 - "n02123045 tabby, tabby cat"
0.2380 - "n02123159 tiger cat"
0.1235 - "n02124075 Egyptian cat"
0.1005 - "n02119022 red fox, Vulpesvulpes"
0.0716 - "n02127052 lynx, catamount"
```

可见给出了 5 个预测结果，按照概率分布从高到低的顺序排列。这种预测结果称为 Top-5 预测结果，对当前样本而言，分类正确率为 5 项之和 0.7768。除 Top-5 预测结果之外，还有 Top-3、Top-1 等预测结果，对当前样本的分类正确率分别为 0.6747、0.3132。我们在第 13 天内容中还会见到它们。

分类准确率不仅与验证数据集有关，与模型的关系也非常密切。我们在 Caffe Model Zoo 上找到几个模型在 ILSVRC 2012 验证数据集上的分类效果，如表 10-1 所示。

表 10-1 分类效果

Name	Top-1	Top-5
BVLC AlexNet ^[1]	0.569	0.801
BVLC CaffeNet	0.571	0.801
NiN ^[2]	0.567	0.795
VGG-DeviS	0.612	0.834
VGG-19 ^[3]	0.685	0.885
BVLC GoogLeNet ^[4]	0.687	0.89
Princeton GoogLeNet	0.672	0.881

可见单模型分类性能最好的是 BVLC GoogLeNet。

通过掌握今天的内容，并学习其他更多深度学习模型的设计和训练方法，天马行空的大胆想象加上对 Caffe 模型的熟练运用，将会让你成长为优秀的模型设计师。

10.5 练习题

1. Caffe 将模型设计与代码实现分开有什么好处？
2. 如果只有 Caffe 训练好的模型权值文件 (*.caffemodel)，而没有求解器状态文件 (*.solverstate)，能否让网络继续训练？能否对网络精调？

10.6 参考资料

[1] Alex Krizhevsky, ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012

[2] Shuicheng Yan, Network In Network, arXiv:1312.4400v3

[3] Very Deep Convolutional Networks for Large-Scale Image Recognition, arXiv:1409.1556v3

[4] Going deeper with convolutions, arXiv:1409.4842v1

第 11 天

Caffe 前向传播计算

使用传统的 BP 算法进行 CNN 训练时包括两个阶段：前向传播计算（Forward）和反向传播计算（Backward）。今天我们将注意力放在前向传播阶段。

前向传播阶段在实际应用中最常见，比如大量的在线系统（语音识别、文字识别、图像分类和检索等）都是仅前向传播阶段的应用；一些嵌入式系统（视觉机器人、无人机、智能语音机器人）受限于计算资源，仅实现前向传播阶段，而反向传播计算则由计算性能更强大的服务器完成。

11.1 前向传播的特点

在前向传播阶段，数据源起于数据读取层，经过若干处理层，到达最后一层（可能是损失层或特征层）。

网络中的权值在前向传播阶段不发生变化，可以看作常量。

网络路径是一个有向无环图（Directed AcyclineGraph, DAG）。从最初的节点出发，经历若干处理层，不存在循环结构，因此数据流会一直向前推进到达终点。

我们可以使用数据流分析方法对前向传播过程进行研究：

从输入数据集中取一个样本 (X, Y) ，其中 X 为数据， Y 为标签。将 X 送入网络，逐层计算，得到相应的网络处理输出 O 。网络执行的计算可以用公式表达为：

$$O=F_n(\dots(F_2(F_1(XW_1)W_2)\dots)W_n)$$

其中， $F_i, i=1,2,\dots,n$ 表示非线性变换，而 $W_i, i=1,2,\dots,n$ 表示各个权值层权值。

得到网络输出 O 后，可以用 (Y, O) 评估网络质量。理想的网络满足 $Y=O$ 。

11.2 前向传播的实现

在 Caffe 中 CNN 前向传播过程由 Net + Layer 组合完成，中间结果和最终结果则使用 Blob 承载。下面我们深入代码来观察这一过程。

11.2.1 DAG 构造过程

首先我们从 Net 构造函数开始。

```
// 从 NetParameter 对象构造
template<typename Dtype>
Net<Dtype>::Net(const NetParameter& param, const Net* root_net)
    : root_net_(root_net) {
    Init(param);
}

// 从 net.prototxt 文件构造
template<typename Dtype>
Net<Dtype>::Net(const string& param_file, Phase phase, const Net* root_net)
    : root_net_(root_net) {
    NetParameter param;
    ReadNetParamsFromTextFileOrDie(param_file, &param);
    param.mutable_state()->set_phase(phase);
    Init(param);
}
```

从上面的构造函数看到，二者都调用了 Init() 函数。传递给该函数的参数 param 是 NetParameter 对象，我们已经在第 8 天的例程中使用过，了解过其数据结构描述 (caffe.proto)。我们可以从 net.prototxt 文件读取到内存中，初始化一个 NetParameter 对象，然后传递给 Init() 函数。

接着追踪 Init() 函数：

```
// 这个函数很长
template<typename Dtype>
void Net<Dtype>::Init(const NetParameter& in_param) {
    CHECK(Caffe::root_solver() || root_net_)
```

```

<< "root_net_ needs to be set for all non-root solvers";
// 根据 NetParameter 对象设置处理阶段 (Train/Test)
phase_ = in_param.state().phase();
NetParameter filtered_param;
FilterNet(in_param, &filtered_param); // 过滤一些参数, 仅保留当前阶段参数
LOG_IF(INFO, Caffe::root_solver())
<< "Initializing net from parameters: "<<std::endl
<<filtered_param.DebugString();
// 创建一个拷贝, 以后就用它了
NetParameter param;
InsertSplits(filtered_param, &param);
// 构建所有 Layer 并将它们连接
name_ = param.name(); // 网络名
map<string, int> blob_name_to_idx; // Blob 名与索引的映射
set<string> available_blobs; // 已有 Blob 名集合
memory_used_ = 0; // 统计内存占用
// 对每个 Layer 设置输入 Blob (BottomBlob) 和输出 Blob (TopBlob)
bottom_vecs_.resize(param.layer_size()); // 有多少层, 就有多少个输入 Blob
top_vecs_.resize(param.layer_size()); // 有多少层, 就有多少个输出 Blob
bottom_id_vecs_.resize(param.layer_size()); // 记录每个层的输入 Blob 索引
param_id_vecs_.resize(param.layer_size()); // 记录每个层的权值 Blob 索引
top_id_vecs_.resize(param.layer_size()); // 记录每个层的输出 Blob 索引
bottom_need_backward_.resize(param.layer_size());
// 记录每个 Blob 是否需要反向传播过程
for (int layer_id = 0; layer_id < param.layer_size(); ++layer_id) {
    // 遍历每个层
    bool share_from_root = !Caffe::root_solver()
    && root_net_ -> layers_[layer_id] -> ShareInParallel();
    // 判断该层是否设置为与其他网络共享
    if (!param.layer(layer_id).has_phase()) {
        // 每个层的阶段标记, 如果在层描述中未指定阶段, 就使用 Net 的阶段
        param.mutable_layer(layer_id) -> set_phase(phase_);
    }
    // 获取层参数
    const LayerParameter& layer_param = param.layer(layer_id);
    // Layer 工厂, 专业制造各种 Layer, 然后添加到 Net 类的 layers_ 对象中
    // 注意到这些 Layer 的 LayerParameter 都继承自 NetParameter
    layers_.push_back(LayerRegistry<Dtype>::CreateLayer(layer_param));
    // 将 Layer 名称添加到 Net 类的 layer_names_ 对象中

```

```

layer_names_.push_back(layer_param.name());
LOG_IF(INFO, Caffe::root_solver())
<< "Creating Layer " << layer_param.name();
bool need_backward = false; // 判断该层是否需要反向传播
// 确定该 Layer 的输入 Blob 和输出 Blob
for (int bottom_id = 0; bottom_id < layer_param.bottom_size();
    ++bottom_id) {
    // 遍历所有输入 Blob，记录到 Blob 名集合、Blob 名到索引映射中
    const int tblob_id = AppendBottom(param, layer_id, bottom_id,
&available_blobs, &blob_name_to_idx);
    // 只要有一个输入 Blob 需要反向传播，那么该层就需要反向传播
    need_backward |= blob_need_backward_[blob_id];
}
// 输出 Blob 做同样的事情
int num_top = layer_param.top_size();
for (int top_id = 0; top_id < num_top; ++top_id) {
    AppendTop(param, layer_id, top_id, &available_blobs, &blob_name_to_idx);
    // 收集输入层 (InputLayer) 信息，如果有，其输出 blob 将作为整个 Net 的输入
    if (layer_param.type() == "Input") {
        const int blob_id = blobs_.size() - 1;
        net_input_blob_indices_.push_back(blob_id);
        net_input_blobs_.push_back(blobs_[blob_id].get());
    }
}
// Layer 连接设置完毕，调用各个 Layer 的 SetUp() 函数
layers_[layer_id]->SetUp(bottom_vecs_[layer_id], top_vecs_[layer_id]);
LOG_IF(INFO, Caffe::root_solver())
<< "Setting up " << layer_names_[layer_id];
// 设置输出 Blob 对损失函数的投票因子
for (int top_id = 0; top_id < top_vecs_[layer_id].size(); ++top_id) {
    if (blob_loss_weights_.size() <= top_id_vecs_[layer_id][top_id]) {
        blob_loss_weights_.resize(top_id_vecs_[layer_id][top_id] + 1, Dtype(0));
    }
    blob_loss_weights_[top_id_vecs_[layer_id][top_id]] = layer->loss(top_id);
    // 打印每层输出 Blob 尺寸信息
    LOG_IF(INFO, Caffe::root_solver())
    << "Top shape: " << top_vecs_[layer_id][top_id]->shape_string();
    if (layer->loss(top_id)) {
        LOG_IF(INFO, Caffe::root_solver())

```



```

    << "    with loss weight "<< layer->loss(top_id);
    // 信不信由你，除了损失层的 loss_weight 为 1，其他层都是 0
}
// 统计每个输出 Blob 内存占用量
memory_used_ += top_vecs_[layer_id][top_id]->count();
}
// 打印所有输出 Blob 内存占用量
LOG_IF(INFO, Caffe::root_solver())
<< "Memory required for data: "<<memory_used_ * sizeof(Dtype);
// 下面开始初始化各层权值 Blob
const int param_size = layer_param.param_size();
const int num_param_blobs = layers_[layer_id]->blobs().size();
// 保证参数配置需要的权值 Blob 数目不大于实际对象的权值 Blob 数目
CHECK_LE(param_size, num_param_blobs)
<< "Too many params specified for layer "<< layer_param.name();
ParamSpec default_param_spec;
// 每个权值层（卷积层、全连接层）都要经历下面的过程
for (int param_id = 0; param_id < num_param_blobs; ++param_id) {
    const ParamSpec* param_spec = (param_id < param_size) ?
    &layer_param.param(param_id) : &default_param_spec;
    const bool param_need_backward = param_spec->lr_mult() != 0;
    // 设置权值层 param(lr_mult: 0) 可以禁止其反向传播过程，即冻结权值
    need_backward |= param_need_backward;
    layers_[layer_id]->set_param_propagate_down(param_id,
    param_need_backward);
}
for (int param_id = 0; param_id < num_param_blobs; ++param_id) {
    // 记录权值 Blob 到 Net 后台数据库
    AppendParam(param, layer_id, param_id);
}
// 最后设置反向传播标志
layer_need_backward_.push_back(need_backward);
if (need_backward) {
    for (int top_id = 0; top_id < top_id_vecs_[layer_id].size(); ++top_id) {
        blob_need_backward_[top_id_vecs_[layer_id][top_id]] = true;
    }
}
}
}
// 略去了一些目前不关注的信息，比如 loss 设置

```

```

// 所有剩下的 Blob 都被看作输出 Blob
for (set<string>::iterator it = available_blobs.begin();
     it != available_blobs.end(); ++it) {
    LOG_IF(INFO, Caffe::root_solver())
    << "This network produces output " << *it;
    net_output_blobs_.push_back(blobs_[blob_name_to_idx[*it]].get());
    net_output_blob_indices_.push_back(blob_name_to_idx[*it]);
}
// 将 Blob 名称与 Blob id 对应关系登记到 Net 后台数据库
for (size_t blob_id = 0; blob_id < blob_names_.size(); ++blob_id) {
    blob_names_index_[blob_names_[blob_id]] = blob_id;
}
// 将 Layer 名称与 Layer id 对应关系登记到 Net 后台数据库
for (size_t layer_id = 0; layer_id < layer_names_.size(); ++layer_id) {
    layer_names_index_[layer_names_[layer_id]] = layer_id;
}
ShareWeights();
debug_info_ = param.debug_info();
LOG_IF(INFO, Caffe::root_solver()) << "Network initialization done.";
}

```

可见，Init()函数完成了非常关键的网络初始化和层初始化操作。虽然代码很长，但我们只要抓住几个核心对象（在表 8-1 中列出），了解其功能并密切关注其动态，即可掌握 Init()函数的执行流程和具体含义。在该函数中，调用了 3 个登记注册函数，我们继续深入阅读其代码：

```

// 登记每层输出 Blob
template<typename Dtype>
void Net<Dtype>::AppendTop(const NetParameter& param, const int layer_id,
                          const int top_id, set<string>* available_blobs,
                          map<string, int>* blob_name_to_idx) {
    shared_ptr<LayerParameter> layer_param(
        new LayerParameter(param.layer(layer_id)));
    const string& blob_name = (layer_param->top_size() > top_id) ?
        layer_param->top(top_id) : "(automatic)";
    // 检测是否为原位计算
    if (blob_name_to_idx->find(layer_param->bottom_size() > top_id) &&
        blob_name == layer_param->bottom(top_id)) {
        // 是原位计算
        LOG_IF(INFO, Caffe::root_solver())
        << layer_param->name() << " -> " << blob_name << " (in-place)";
    }
}

```

```

top_vecs_[layer_id].push_back(blobs_[(*blob_name_to_idx)[blob_name]].get());
top_id_vecs_[layer_id].push_back((*blob_name_to_idx)[blob_name]);
} else if (blob_name_to_idx&&
           blob_name_to_idx->find(blob_name) != blob_name_to_idx->end()) {
    // 如果不是原位计算, 但名字重复, 则报错
    LOG(FATAL) << "Top blob '" << blob_name
<< "' produced by multiple sources.";
} else {
    // 正常输出
    if (Caffe::root_solver()) {
        LOG(INFO) << layer_param->name() << " -> " << blob_name;
    }
    shared_ptr<Blob<Dtype>> blob_pointer(new Blob<Dtype>());
    // 新建一个 Blob, 插入到 Net::blobs_ 最后
    const int blob_id = blobs_.size();
    blobs_.push_back(blob_pointer);
    blob_names_.push_back(blob_name);
    blob_need_backward_.push_back(false);
    if (blob_name_to_idx) { (*blob_name_to_idx)[blob_name] = blob_id; }
    top_id_vecs_[layer_id].push_back(blob_id);
    top_vecs_[layer_id].push_back(blob_pointer.get());
}
if (available_blobs) { available_blobs->insert(blob_name); }
}

// 登记每层输入 Blob
template<typename Dtype>
int Net<Dtype>::AppendBottom(const NetParameter& param, const int layer_id,
                           const int bottom_id, set<string>* available_blobs,
                           map<string, int>* blob_name_to_idx) {
    const LayerParameter& layer_param = param.layer(layer_id);
    const string& blob_name = layer_param.bottom(bottom_id);
    if (available_blobs->find(blob_name) == available_blobs->end()) {
        LOG(FATAL) << "Unknown bottom blob '" << blob_name << "' (layer '"
<< layer_param.name() << "', bottom index " << bottom_id << ")";
    }
    const int blob_id = (*blob_name_to_idx)[blob_name];
    LOG_IF(INFO, Caffe::root_solver())
<< layer_names_[layer_id] << " <- " << blob_name;

```

```

bottom_vecs_[layer_id].push_back(blobs_[blob_id].get());
bottom_id_vecs_[layer_id].push_back(blob_id);
available_blobs->erase(blob_name);
boolpropagate_down = true;
// 检查是否可以跳过反向传播
if (layer_param.propagate_down_size() > 0)
    propagate_down = layer_param.propagate_down(bottom_id);
const bool need_backward = blob_need_backward_[blob_id] && propagate_down;
bottom_need_backward_[layer_id].push_back(need_backward);
return blob_id;
}
// 登记每层权值 Blob
template<typename Dtype>
void Net<Dtype>::AppendParam(const NetParameter& param, const int layer_id,
                             const int param_id) {
    const LayerParameter& layer_param = layers_[layer_id]->layer_param();
    const int param_size = layer_param.param_size();
    stringparam_name =
        (param_size>param_id) ? layer_param.param(param_id).name() : "";
    if (param_name.size()) {
        param_display_names_.push_back(param_name);
    } else {
        ostringstreamparam_display_name;
        param_display_name<<param_id;
        param_display_names_.push_back(param_display_name.str());
    }
    const int net_param_id = params_.size();
    params_.push_back(layers_[layer_id]->blobs()[param_id]);
    param_id_vecs_[layer_id].push_back(net_param_id);
    param_layer_indices_.push_back(make_pair(layer_id, param_id));
    ParamSpecdefault_param_spec;
    const ParamSpec* param_spec = (layer_param.param_size() > param_id) ?
    &layer_param.param(param_id) : &default_param_spec;
    if (!param_size || !param_name.size() || (param_name.size() &&
        param_names_index_.find(param_name) == param_names_index_.end())) {
        // 该层拥有权值 Blob
        param_owners_.push_back(-1);
        if (param_name.size()) {
            param_names_index_[param_name] = net_param_id;

```

```

}

const int learnable_param_id = learnable_params_.size();
learnable_params_.push_back(params_[net_param_id].get());
learnable_param_ids_.push_back(learnable_param_id);
has_params_lr_.push_back(param_spec->has_lr_mult());
has_params_decay_.push_back(param_spec->has_decay_mult());
params_lr_.push_back(param_spec->lr_mult());
params_weight_decay_.push_back(param_spec->decay_mult());
} else {
    // 该层共享权值 Blob
    const int owner_net_param_id = param_names_index_[param_name];
    param_owners_.push_back(owner_net_param_id);
    const pair<int, int>&owner_index =
        param_layer_indices_[owner_net_param_id];
    const int owner_layer_id = owner_index.first;
    const int owner_param_id = owner_index.second;
    LOG_IF(INFO, Caffe::root_solver()) << "Sharing parameters '" <<param_name
<< "' owned by "
<< "layer '" <<layer_names_[owner_layer_id] <<'", param "'
<< "index " <<owner_param_id;
    Blob<Dtype>* this_blob = layers_[layer_id]->blobs()[param_id].get();
    Blob<Dtype>* owner_blob =
        layers_[owner_layer_id]->blobs()[owner_param_id].get();
    const int param_size = layer_param.param_size();
    if (param_size>param_id&& (layer_param.param(param_id).share_mode() ==
        ParamSpec_DimCheckMode_PERMISSIVE)) {
        // 检查允许的维度
        CHECK_EQ(this_blob->count(), owner_blob->count())
<< "Cannot share param '" <<param_name<< "' owned by layer '"
<< layer_names_[owner_layer_id] << "' with layer '"
<< layer_names_[layer_id] << "'; count mismatch. Owner layer param "
<< "shape is " <<owner_blob->shape_string() << " ; sharing layer "
<< "shape is " <<this_blob->shape_string();
    } else {
        // 严格检查允许的维度
        CHECK(this_blob->shape() == owner_blob->shape())
<< "Cannot share param '" <<param_name<< "' owned by layer '"
<< layer_names_[owner_layer_id] << "' with layer '"
<< layer_names_[layer_id] << "'; shape mismatch. Owner layer param "

```

```

<< "shape is "<<owner_blob->shape_string() << "; sharing layer "
<< "expects shape "<<this_blob->shape_string();
}
const int learnable_param_id = learnable_param_ids_[owner_net_param_id];
learnable_param_ids_.push_back(learnable_param_id);
if (param_spec->has_lr_mult()) {
    if (has_params_lr_[learnable_param_id]) {
        CHECK_EQ(param_spec->lr_mult(), params_lr_[learnable_param_id])
<< "Shared param '" <<param_name<< "' has mismatched lr_mult.";
    } else {
        has_params_lr_[learnable_param_id] = true;
        params_lr_[learnable_param_id] = param_spec->lr_mult();
    }
}
if (param_spec->has_decay_mult()) {
    if (has_params_decay_[learnable_param_id]) {
        CHECK_EQ(param_spec->decay_mult(),
            params_weight_decay_[learnable_param_id])
<< "Shared param '" <<param_name<< "' has mismatched decay_mult.";
    } else {
        has_params_decay_[learnable_param_id] = true;
        params_weight_decay_[learnable_param_id] = param_spec->decay_mult();
    }
}
}
}
}

```

11.2.2 Net Forward 实现

掌握了前一节的内容，下面我们很自然地就能看懂 Forward 代码。

```

template<typename Dtype>
Dtype Net<Dtype>::ForwardFromTo(int start, int end) {
// 计算从第 start 到 end 层的前向传播过程
CHECK_GE(start, 0);
CHECK_LT(end, layers_.size());
Dtype loss = 0;
for (int i = start; i <= end; ++i) {
    // LOG(ERROR) << "Forwarding " <<layer_names_[i];
    // 调用每个 Layer 的 Forward() 函数，得到每层 loss

```

```

Dtype layer_loss = layers_[i]->Forward(bottom_vecs_[i], top_vecs_[i]);
loss += layer_loss;
if (debug_info_) { ForwardDebugInfo(i); }
}
// 返回 loss 值
return loss;
}

```

```

template<typename Dtype>
Dtype Net<Dtype>::ForwardFrom(int start) {
    // 计算从 start 开始到最后一层的前向传播过程
    return ForwardFromTo(start, layers_.size() - 1);
}

```

```

template<typename Dtype>
Dtype Net<Dtype>::ForwardTo(int end) {
    // 计算从第一层到第 end 层的前向传播过程
    return ForwardFromTo(0, end);
}

```

```

template<typename Dtype>
const vector<Blob<Dtype>*>& Net<Dtype>::Forward(Dtype* loss) {
    // 计算整个网络前向传播过程，返回损失值（可选）和网络输出 Blob
    if (loss != NULL) {
        *loss = ForwardFromTo(0, layers_.size() - 1);
    } else {
        ForwardFromTo(0, layers_.size() - 1);
    }
    return net_output_blobs_;
}

```

```

template<typename Dtype>
const vector<Blob<Dtype>*>& Net<Dtype>::Forward(
    const vector<Blob<Dtype>*>& bottom, Dtype* loss) {
    // 接受输入 Blob 作为 Net 输入，计算前向传播，得到损失值（可选）和网络输出 Blob
    LOG_EVERY_N(WARNING, 1000) << "DEPRECATED: Forward(bottom, loss) "
    << "will be removed in a future version. Use Forward(loss).";
    // 直接将输入 Blob 拷贝到 net_input_blobs_ 中
    for (int i = 0; i < bottom.size(); ++i) {

```

```
    net_input_blobs_[i]->CopyFrom(*bottom[i]);  
}  
return Forward(loss);  
}
```

根据以上几个不同版本的 Forward 函数，读者应该能够在脑海中形成 DAG 数据流动图。我们明天继续深入反向传播过程。

11.3 练习题

1. 阅读每个 Layer 的 SetUp()函数实现。
2. 阅读每个 Layer 的 Forward_cpu()函数实现。

第 12 天

Caffe 反向传播计算

使用传统的 BP 算法进行 CNN 训练时包括两个阶段：前向传播计算（Forward）和反向传播计算（Backward）。今天我们将注意力放在反向传播阶段。

当我第一次在一台联想笔记本电脑（带独显 Geforce 610M）上将 Caffe 编译成功时，想第一时间运行训练任务，用了一天时间将 ImageNet 数据集转换为 LEVELDB。然而训练十分缓慢，同事对我说，你的笔记本电脑计算能力太低，不妨只运行前向预测任务。后来使用公司服务器（带双 GPU Tesla K80），单机四卡运行训练任务速度提升显著。

反向传播过程只有在训练环境下才需要计算，由于消耗时间较长，对计算资源要求较高，一般为离线服务。

12.1 反向传播的特点

CNN 进行前向传播阶段，依次调用每个 Layer 的 Forward 函数，得到逐层的输出，最后一层与目标函数比较得到损失函数，计算误差更新值，通过反向传播路径逐层到达第一层，所有权值层在反向传播结束后一起更新。

12.2 损失函数

损失层（Loss Layer）是 CNN 的终点，接受两个 Blob 作为输入，其中一个为 CNN 的预测值；另一个是真实标签。损失层则将这两个输入进行一系列运算，得到当前网络的损失函数（Loss Function），一般记为 $L(\theta)$ ，其中 θ 表示当前网络权值构成的向量空间。机器学习的目的是在权值空间中找到让损失函数 $L(\theta)$ 最小的权值 θ_{opt} ，可以采用一系列最优化方法（如后面将会介绍的

SGD 方法)逼近权值 θ_{opt} 。

损失函数是在前向传播计算中得到的,同时也是反向传播的起点。

12.2.1 算法描述

Caffe 中实现了多种损失层,分别用于不同场合。其中 SoftmaxWithLossLayer 实现了 Softmax+交叉熵损失函数计算过程,适用于单 label 的分类问题;另外还有欧式损失函数(用于回归问题)、Hinge 损失函数(最大间隔分类, SVM)、Sigmoid + 交叉熵损失函数(用于多属性/多分类问题)等。今天我们只关注最基本的 SoftmaxWithLossLayer,其他损失层的算法可以直接看 Caffe 相应源码。

假设有 K 个类别, Softmax 计算过程为:

$$\text{Soft max}(a_i) = \frac{\exp(a_i)}{\sum_j \exp(a_j)}, i = 0, 1, 2, \dots, K-1$$

Softmax 的结果相当于输入图像被分到每个标签的概率分布。根据高等数学知识,该函数是单调增函数,即输入值越大,输出也越大,输入图像属于该标签的概率就越大。

对 Softmax 的结果计算交叉熵分类损失函数为:

$$L(\theta) = -\frac{1}{N} \sum_i \log[\text{Soft max}(a_k)], i = 0, 1, 2, \dots, N-1$$

其中, k 为真实标签值, N 为一个批量的大小。

TIPS: Caffe 一些不为人知的数字

理想的分类器应当是除了真实标签的概率为 1,其余标签概率均为 0,这样计算得到其损失函数为 $-\ln(1) = 0$ 。损失函数越大,说明该分类器在真实标签上分类概率越小,性能也就越差。一个非常差的分类器,可能在真实标签上的分类概率接近于 0,那么损失函数就接近于正无穷,我们称为训练发散,需要调小学习速率。在 ImageNet-1000 分类问题中,初始状态为均匀分布,每个类别的分类概率均为 0.001,故此时计算损失函数值为 $-\ln(0.001) = \ln(1000) = 6.907755\dots$ 。经常有同学问,“我的 loss 为什么总是在 6.9 左右(该现象被称为 6.9 高原反应),训练了好久都不下降呢?”说明还都没有训练收敛的迹象,尝试调大学习速率,或者修改权值初始化方式。

12.2.2 参数描述

先看一下 caffe.proto, 找到有关 Softmax 的消息定义:

```
// 用于 SoftmaxLayer, SoftmaxWithLossLayer
message SoftmaxParameter {
    // 计算引擎选择
    enum Engine {
        DEFAULT = 0;
        Caffe = 1;
        CUDNN = 2; // 使用 CUDNN 计算引擎
    }
    optional Engine engine = 1 [default = DEFAULT]; // 默认为 0
    // axis 为可选参数, 指定沿哪个维度计算 Softmax, 可以是负数, 表示从后向前索引
    optional int32 axis = 2 [default = 1];
}
```

12.2.3 源码分析

损失层的基类声明于 include/caffe/loss_layers.hpp 中:

```
// 损失层的鼻祖类, 派生于 Layer
template<typename Dtype>
class LossLayer : public Layer<Dtype> {
public:
    // 显式构造函数
    explicit LossLayer(const LayerParameter& param)
        : Layer<Dtype>(param) {}
    // 层配置函数
    virtual void LayerSetUp(
        const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top);
    // 变形函数
    virtual void Reshape(
        const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top);
    // 接受两个 Blob 作为输入
    virtual inline int ExactNumBottomBlobs() const { return 2; }
    // 为了方便和后向兼容, 指导 Net 为损失层自动分配单个输出 Blob, 损失层则会将计算结果  $L(\theta)$  保存在这里
    virtual inline bool AutoTopBlobs() const { return true; }
```

```

// 只有一个输出 Blob
virtual inline int ExactNumTopBlobs() const { return 1; }
// 我们经常不能对标签做反向传播计算，故忽略 force_backward
virtual inline bool AllowForceBackward(const int bottom_index) const {
    return bottom_index != 1;
}
};

```

用来计算 Softmax 损失函数的层 SoftmaxLayer 声明在 include/caffe/layers/softmax_layer.hpp 中：

```

//SoftmaxLayer 直接派生于 Layer
template<typename Dtype>
class SoftmaxLayer : public Layer<Dtype> {
public:
    // 显式构造函数
    explicit SoftmaxLayer(const LayerParameter& param)
        : Layer<Dtype>(param) {}
    // 变形函数
    virtual void Reshape(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    // 返回类名字串
    virtual inline const char* type() const { return "Softmax"; }
    // 该层接受一个输入 Blob，产生一个输出 Blob
    virtual inline int ExactNumBottomBlobs() const { return 1; }
    virtual inline int ExactNumTopBlobs() const { return 1; }

protected:
    // 前向传播函数
    virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    // 反向传播函数
    virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);
    virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);
    // 计算参数
    int outer_num_;

```

```

int inner_num_;
int softmax_axis_;
//利用 BLAS 计算求和
Blob<Dtype>sum_multiplier_;
//用来临时存放中间结果的 Blob
Blob<Dtype> scale_;
};

```

SoftmaxLayer 实现在 src/caffe/layers/softmax_layer.cpp 中,我们深入内部来看一下具体实现:

// 变形函数

```

template<typename Dtype>
void SoftmaxLayer<Dtype>::Reshape(const vector<Blob<Dtype>*>& bottom, vector<Blob<Dtype>*>*& top)
{
    (*top)[0]->Reshape(bottom[0]->num(), bottom[0]->channels(), bottom[0]->height(),
bottom[0]->width());
    // Softmax 层的输出与输入尺寸大小一致
    sum_multiplier_.Reshape(1, bottom[0]->channels(), 1, 1);
    // sum_multiplier_尺寸: 1 * channels * 1 * 1, 值全为 1
    Dtype* multiplier_data = sum_multiplier_.mutable_cpu_data();
    for (int i = 0; i<sum_multiplier_.count(); ++i)
    {
        multiplier_data[i] = 1.;
    }
    // scale_尺寸: num* 1 * height * width
    scale_.Reshape(bottom[0]->num(), 1, bottom[0]->height(), bottom[0]->width());
}

```

// 前向计算, 得到 Softmax(a_k) 值

```

template<typename Dtype>
void SoftmaxLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom, vector<Blob<Dtype>*>*& top)
{
    // 获得输入/输出 Blob 数据指针
    const Dtype* bottom_data = bottom[0]->cpu_data();
    Dtype* top_data = (*top)[0]->mutable_cpu_data();
    Dtype* scale_data = scale_.mutable_cpu_data();
    int num = bottom[0]->num();
    int channels = bottom[0]->channels();
}

```

```

int dim = bottom[0]->count() / bottom[0]->num(); // 总的类别数目
int spatial_dim = bottom[0]->height() * bottom[0]->width();
// 信不信由你，如果前一层是全连接层，spatial_dim 总为 1
// 先将输入拷贝到输出缓冲区
caffe_copy(bottom[0]->count(), bottom_data, top_data);
// 减去最大值，避免数值问题，计算指数，归一化
for (int i = 0; i < num; ++i)
{
    // 初始化 scale_ 的 data 域为第一个平面
    caffe_copy(spatial_dim, bottom_data + i * dim, scale_data);
    for (int j = 0; j < channels; j++)
    {
        for (int k = 0; k < spatial_dim; k++)
        {
            scale_data[k] = std::max(scale_data[k], bottom_data[i * dim + j * spatial_dim + k]);
        }
    }
    // 输出缓冲区减去最大值 a_k = a_k - max(a_i)
    caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, channels, spatial_dim, 1, -1.,
sum_multiplier_.cpu_data(), scale_data, 1., top_data + i * dim);
    // 求指数项 exp(a_k)
    caffe_exp<Dtype>(dim, top_data + i * dim, top_data + i * dim);
    // 求 1 + exp(a_k)
    caffe_cpu_gemv<Dtype>(CblasTrans, channels, spatial_dim, 1., top_data + i * dim,
sum_multiplier_.cpu_data(), 0., scale_data);
    // 求 Softmax 值，即 exp(a_k) / (1 + exp(a_k))
    for (int j = 0; j < channels; j++)
    {
        caffe_div(spatial_dim, top_data + (*top)[0]->offset(i, j), scale_data, top_data +
(*top)[0]->offset(i, j));
    }
}
}

// Softmax 的反向传播函数，要学会求导法则
template<typename Dtype>
void SoftmaxLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*> & top,
    const vector<bool> & propagate_down,
    vector<Blob<Dtype>*> * bottom)
{

```

```

// 获得 data、diff 指针
const Dtype* top_diff = top[0]->cpu_diff();
const Dtype* top_data = top[0]->cpu_data();
Dtype* bottom_diff = (*bottom)[0]->mutable_cpu_diff();
Dtype* scale_data = scale_.mutable_cpu_data();
int num = top[0]->num();
int channels = top[0]->channels();
int dim = top[0]->count() / top[0]->num();
int spatial_dim = top[0]->height() * top[0]->width();
caffe_copy(top[0]->count(), top_diff, bottom_diff);
for (int i = 0; i < num; ++i) {
    // 计算 top_diff 和 top_data 的点积, 然后从 bottom_diff 中减去该值
    for (int k = 0; k < spatial_dim; ++k)
    {
        scale_data[k] = caffe_cpu_strided_dot<Dtype>(channels,
            bottom_diff + i * dim + k, spatial_dim,
            top_data + i * dim + k, spatial_dim);
    }
    // 减值
    caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, channels, spatial_dim, 1,
        -1., sum_multiplier_.cpu_data(), scale_data, 1., bottom_diff + i * dim);
}
// 逐点相乘
caffe_mul(top[0]->count(), bottom_diff, top_data, bottom_diff);
}

```

看完了 Softmax 函数的计算过程, 那么损失层 SoftmaxWithLossLayer 的实现就非常简单了, 找到 SoftmaxWithLossLayer 的声明如下:

```

// 前置声明 SoftmaxLayer, 便于 SoftmaxWithLossLayer 使用
template<typename Dtype> class SoftmaxLayer;
// 将实数预测向量通过 Softmax 计算获得每个类别的概率分布
// 这个类比单独 SoftmaxLayer + MultinomialLogisticLossLayer 在梯度数值计算上更加稳定
// Test 阶段, 这个层可以直接用 SoftmaxLayer 代替
/**
 * 输入 Blob 1 为预测结果, 形状为 N x K x 1 x 1, K 为总类别数目, N 为批量数。取值范围为 (-Inf, Inf),
 * 表示每个类别获得的分类 score, 值越大说明输入图像与该类别越接近
 * 输入 Blob 2 为真实标签, 形状为 N x 1 x 1 x 1
 * 输出 Blob 为计算得到的交叉熵分类损失 E, 形状为 1 x 1 x 1 x 1
 */

```

```

template<typename Dtype>
class SoftmaxWithLossLayer : public LossLayer<Dtype> {
public:
    explicit SoftmaxWithLossLayer(const LayerParameter& param)
        : LossLayer<Dtype>(param) {}
    virtual void LayerSetUp(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    virtual void Reshape(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);

    virtual inline const char* type() const { return "SoftmaxWithLoss"; }
    virtual inline int ExactNumTopBlobs() const { return -1; }
    virtual inline int MinTopBlobs() const { return 1; }
    virtual inline int MaxTopBlobs() const { return 2; }

protected:
    /// @copydoc SoftmaxWithLossLayer
    virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);
    virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);

    /// 内置一个 SoftmaxLayer 对象
    shared_ptr<Layer<Dtype>> softmax_layer_;
    /// prob_ 用于存储 SoftmaxLayer 计算的分类预测结果
    Blob<Dtype> prob_;
    ///
    vector<Blob<Dtype>*> softmax_bottom_vec_;
    ///
    vector<Blob<Dtype>*> softmax_top_vec_;
    ///
    bool has_ignore_label_;
    ///
    int ignore_label_;

```



```
bool normalize_;

int softmax_axis_, outer_num_, inner_num_;
};
```

该层的 `SetUp` 过程如下：

```
template<typename Dtype>
void SoftmaxWithLossLayer<Dtype>::LayerSetUp(
    const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top) {
    LossLayer<Dtype>::LayerSetUp(bottom, top);
    // 创建时动态修改本层的 LayerParameter 参数，适应 SoftmaxLayer
    LayerParametersoftmax_param(this->layer_param_);
    softmax_param.set_type("Softmax");
    // 创建 SoftmaxLayer
    softmax_layer_ = LayerRegistry<Dtype>::CreateLayer(softmax_param);
    softmax_bottom_vec_.clear();
    softmax_bottom_vec_.push_back(bottom[0]);
    softmax_top_vec_.clear();
    softmax_top_vec_.push_back(&prob_);
    softmax_layer_->SetUp(softmax_bottom_vec_, softmax_top_vec_);
    // 剩下的不关心……
}
```

可见，在 `SetUp` 阶段，创建了内部 `SoftmaxLayer` 对象并配置了其输入/输出 `Blob`，然后调用该对象的 `SetUp` 函数。

下面看看 `SoftmaxWithLossLayer` 的前向传播函数：

```
// 前向传播
template<typename Dtype>
void SoftmaxWithLossLayer<Dtype>::Forward_cpu(
    const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top) {
    // 内部 SoftmaxLayer 的前向传播计算
    softmax_layer_->Forward(softmax_bottom_vec_, softmax_top_vec_);
    // 获得概率密度
    const Dtype* prob_data = prob_.cpu_data();
    // 获得标签值
    const Dtype* label = bottom[1]->cpu_data();
    int dim = prob_.count() / outer_num_;
    int count = 0;
```

```

Dtype loss = 0;
for (int i = 0; i < outer_num_; ++i) {
    for (int j = 0; j < inner_num_; ++j) {
        const int label_value = static_cast<int>(label[i * inner_num_ + j]);
        if (has_ignore_label_ && label_value == ignore_label_) {
            continue;
        }
        DCHECK_GE(label_value, 0);
        DCHECK_LT(label_value, prob_.shape(softmax_axis_));
        // 计算损失函数 -log(prob[label])
        loss -= log(std::max(prob_data[i * dim + label_value * inner_num_ + j],
Dtype(FLT_MIN)));
        ++count;
    }
}
// 设置输出 Blob 值
top[0]->mutable_cpu_data()[0] = loss / get_normalizer(normalization_, count);
if (top.size() == 2) {
    top[1]->ShareData(prob_);
}
}

```

可见通过内部 SoftmaxLayer 对象非常简洁。我们再看一下 Backward 计算：

```

template<typename Dtype>
void SoftmaxWithLossLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom) {
    if (propagate_down[1]) {
        // label 输入 Blob 不做反向传播
        LOG(FATAL) << this->type()
<< " Layer cannot backpropagate to label inputs.";
    }
    if (propagate_down[0]) {
        Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
        const Dtype* prob_data = prob_.cpu_data();
        // 将概率密度拷贝到输入 Blob 的 diff 域
        caffe_copy(prob_.count(), prob_data, bottom_diff);
        const Dtype* label = bottom[1]->cpu_data();
        int dim = prob_.count() / outer_num_;
        int count = 0;
    }
}

```

```

for (int i = 0; i < outer_num_; ++i) {
    for (int j = 0; j < inner_num_; ++j) {
        const int label_value = static_cast<int>(label[i * inner_num_ + j]);
        if (has_ignore_label_ && label_value == ignore_label_) {
            for (int c = 0; c < bottom[0]->shape(softmax_axis_); ++c) {
                bottom_diff[i * dim + c * inner_num_ + j] = 0;
            }
        } else {
            // 在输入 Blob 的 diff 域, 计算当前概率密度与理想概率密度 (label 对应类别概率为 1, 其他类别
            // 概率为 0) 之差, 实现误差反向传播
            bottom_diff[i * dim + label_value * inner_num_ + j] -= 1;
            ++count;
        }
    }
}

// 适当缩放
 DtypeLoss_weight = top[0]->cpu_diff()[0] /
                    get_normalizer(normalization_, count);
caffe_scal(prob_.count(), loss_weight, bottom_diff);
}
}

```

通过对 Caffe 损失层的研究, 我们了解到, 前向传播阶段数据逐层传播, 到损失层计算预测概率密度和损失函数; 而反向传播阶段则从损失层开始, 由预测概率密度与理想概率密度 (这就是有监督学习的佐证) 差值得到误差 (diff), 然后将由下一节内容逐层反向传播。我们已经知道一个 Blob 是由 data 和 diff 两部分构成的, 如果说数据读取层是 data 之源, 那么损失层就是 diff 之源。

12.3 反向传播的实现

我们接上昨天介绍的内容, 继续深入研究 Caffe Net 数据结构中的 Backward 函数。

```

// 从第 start 层反向传播到达第 end 层, start >= end
template<typename Dtype>
void Net<Dtype>::BackwardFromTo(int start, int end) {
    CHECK_GE(end, 0);
    CHECK_LT(start, layers_.size());
    for (int i = start; i >= end; --i) {
        if (layer_need_backward_[i]) {

```

```

// 遍历每个层，调用相应的 Backward 函数
layers_[i]->Backward(
    top_vecs_[i], bottom_need_backward_[i], bottom_vecs_[i]);
if (debug_info_) { BackwardDebugInfo(i); }
}
}

// 从第 start 层开始到第一层的反向传播过程
template<typename Dtype>
void Net<Dtype>::BackwardFrom(int start) {
    BackwardFromTo(start, 0);
}

// 从最后一层开始到第 end 层的反向传播过程
template<typename Dtype>
void Net<Dtype>::BackwardTo(int end) {
    BackwardFromTo(layers_.size() - 1, end);
}

// 整个网络的反向传播过程
template<typename Dtype>
void Net<Dtype>::Backward() {
    BackwardFromTo(layers_.size() - 1, 0);
    if (debug_info_) {
        // 如果打开了调试信息开关（在 prototxt 中设定），则计算所有权值的 data/diff 的 L1、L2 范数，监控
        其变化情况，避免发散
        Dtype asum_data = 0, asum_diff = 0, sumsq_data = 0, sumsq_diff = 0;
        for (int i = 0; i < learnable_params_.size(); ++i) {
            asum_data += learnable_params_[i]->asum_data();
            asum_diff += learnable_params_[i]->asum_diff();
            sumsq_data += learnable_params_[i]->sumsq_data();
            sumsq_diff += learnable_params_[i]->sumsq_diff();
        }
        const Dtype l2norm_data = std::sqrt(sumsq_data);
        const Dtype l2norm_diff = std::sqrt(sumsq_diff);
        LOG(ERROR) << "    [Backward] All net params (data, diff): "
        << "L1 norm = (" << asum_data << ", " << asum_diff << "); "
        << "L2 norm = (" << l2norm_data << ", " << l2norm_diff << ");"
    }
}

// 更新权值函数，在反向传播结束后调用
template<typename Dtype>

```

```

void Net<Dtype>::Update() {
    for (int i = 0; i < learnable_params_.size(); ++i) {
        // 调用内部 Blob 的 Update() 函数，具体计算为 data = data - diff，参考第 8 天的例程
        learnable_params_[i]->Update();
    }
}

// 权值 diff 清零
template<typename Dtype>
void Net<Dtype>::ClearParamDiffs() {
    for (int i = 0; i < learnable_params_.size(); ++i) {
        Blob<Dtype>* blob = learnable_params_[i];
        switch (Caffe::mode()) {
            case Caffe::CPU:
                caffe_set(blob->count(), static_cast<Dtype>(0),
                    blob->mutable_cpu_diff());
                break;
            case Caffe::GPU:
#ifdef CPU_ONLY
                caffe_gpu_set(blob->count(), static_cast<Dtype>(0),
                    blob->mutable_gpu_diff());
#else
                NO_GPU;
#endif
                break;
        }
    }
}

```

我们通过今天的内容，知道了 Caffe 反向传播的来龙去脉，对于设计更复杂的有监督学习算法具有指导意义。读者可以深入阅读其他机器学习理论书籍或学术论文，并尝试利用 Caffe 实现。经历这样一个过程，你会成为优秀的机器学习算法工程师。

12.4 练习题

1. Caffe 在计算 Softmax 时，如何避免数值不稳定？
2. 如果一个机器人对 N 分类问题采用随机猜测的方式分类，假设猜对每个类别的概率都是 $1/N$ ，那么 Top-K 准确率 ($K < N$) 应该为多少？

3. 如果用 Caffe 做 ImageNet-22K 分类问题，损失层采用 SoftmaxWithLoss，那么初始状态的损失函数值应该是多少？Cifar-10 和 Cifar-100 呢？
4. 阅读每个 Layer 中 Backward_cpu()函数的实现。
5. 思考 Blob 中的 data 和 diff 能否合并为一个？

第 13 天

Caffe 最优化求解过程

将前面几天讲述的不同 Layer 组合起来就可以构建起完整的 CNN/DNN。今天将从 Caffe 的 Solver 类入手，对 Caffe 训练时的流程做深入分析，也就是看 Caffe 实际是如何“动”起来的。

13.1 求解器是什么

从前两天内容我们学习到，Net 已经完成部分学习任务（数据前向传播、误差反向传播）。而 CNN 剩下有监督的优化过程、利用每层的梯度生成权值增量则由求解器（Solver）负责。

求解器负责对模型优化，它的 KPI（Key Performance Indicator，关键绩效指标，某些公司常用的一种员工绩效评定方式）就是让损失函数达到全局最小。

求解器的特性如下：

- 负责记录优化过程，创建用于学习的训练网络和用于评估学习效果的测试网络。
- 调用 Forward→调用 Backward→更新权值，反复迭代优化模型。
- 周期性地评估测试网络。
- 在优化过程中为模型、求解器状态打快照。

为了让权值从初始化状态向着更好的模型前进，求解器在每次迭代中做了如下事情：

- 调用 Net 的前向传播函数来计算输出和损失函数。
- 调用 Net 的反向传播函数来计算梯度。

- 根据求解器方法，将梯度转换为权值增量。
- 根据学习速率、历史权值、所用方法更新求解器状态。

13.2 求解器是如何实现的

在卷积神经网络中，有两种类型的层需要学习：卷积层和全连接层（统称权值层，因为它们都有 weight 参数）。在设计求解器时，学习速率参数的设置也是针对这两个层的。

我们都知道两个成语：隔靴搔痒和矫枉过正，也就是做事要把握“度”。在 Caffe 中，这个“度”就是控制收敛的参数——学习速率。

我们来具体看一下 Caffe 是如何对权值学习做到“不偏不倚”的。

13.2.1 算法描述

Caffe 中的求解器有以下几种：

- 随机梯度下降法（Stochastic Gradient Descent, SGD），最常用的一种
- AdaDelta
- 自适应梯度法（Adaptive Gradient, ADAGRAD）
- Adam
- Nesterov 加速梯度法（Nesterov's Accelerated Gradient, NAG）
- RMSprop

求解器方法重点是最小化损失函数的全局优化问题，对于数据集 D ，优化目标是在全数据集 D 上损失函数平均值：

$$L(W) = \frac{1}{|D|} \sum_i^{ |D| } f_w(X^{(i)}) + \lambda r(W)$$

其中， $f_w(X^{(i)})$ 是在数据实例 $X^{(i)}$ 上的损失函数， $r(W)$ 为规整项， λ 为规整项的权重。数据集 D 可能非常大，工程上一般在每次迭代中使用这个目标函数的随机逼近，即小批量数据 $N \ll |D|$ 个数据实例：

$$L(W) \approx \frac{1}{N} \sum_i^N f_w(X^{(i)}) + \lambda r(W)$$

模型在前向传播计算损失函数 f_w ，反向传播计算梯度 ∇f_w 。权值增量 ΔW 由求解器通过误差梯度 ∇f_w 、规整项梯度 $\nabla r(W)$ ，以及其他与方法相关的项求解得到。

随机梯度下降法（SGD）利用负梯度 $\nabla L(W)$ 和权值更新历史 V_t 的线性组合更新权值 W 。

学习速率（Learning Rate, LR） α 是负梯度的权重，遗忘因子（Momentum） μ 是权值更新历史 V_t 的权重。顾名思义，学习速率和遗忘因子分别代表今天学了多少新知识，以及以前学过的知识还记得多少。有些同学学习新知识很快，忘得也快；而有些同学虽然学得慢，但记得牢，在这里是一个道理。

已知第 t 次迭代的 V_t 和权值 W_t ，用如下公式可以计算出第 $t+1$ 次迭代的 V_{t+1} 和 W_{t+1} ：

$$\begin{aligned} V_{t+1} &= \mu V_t - \alpha \nabla L(W_t) \\ W_{t+1} &= W_t + V_{t+1} \end{aligned}$$

为了达到最优学习效果，学习过程的“超参数”（hyperparameter，这里为 α 和 μ ）需要仔细调节，其指导原则为：

（1）初始学习速率 $\alpha \approx 0.01 = 10^{-2}$ 。经过迭代训练，当损失函数不再下降时，对 α 进行固定常数衰减（如乘上 0.1），如此反复贯穿训练始终。

（2）遗忘因子 $\mu = 0.9$ 。遗忘因子可以随着迭代对权值增量进行平滑，这样可以让基于 SGD 算法的深度学习系统更稳定、收敛更快。

下面例子是 Krizhevsky 在 ILSVRC 2012 比赛中获胜的学习策略。在 Caffe 中实现该策略非常简单，只需在 solver.prototxt 中写入如下内容：

```
base_lr: 0.01           // 基准学习速率  $\alpha=0.01$ ，另外每个 Layer 会在基准上进行细调
lr_policy: "step"       // 学习速率衰减策略，step 为步进方式，即每进行 step 次迭代，学习速率更新一次
gamma: 0.1              // 学习速率衰减常数，每次更新学习速率都是乘上这个固定常数
stepsize: 100000        // 每 10 万次迭代，对学习速率进行一次更新
max_iter: 350000        // 训练总共需要 35 万次迭代
momentum: 0.9           // 遗忘因子  $\mu=0.9$ 
```

注意：如果训练发散（损失函数值变得非常大，甚至出现 NaN 或 Inf），试着减小 base_lr，然后重新训练，直到不再发散。

13.2.2 数据结构描述

依然先打开 `caffe.proto`，查看与 Solver 相关的描述信息：

```
message SolverParameter {
  // 指定 Net 结构描述文件 (*.prototxt)
  optional string net = 24;
  // 内置 Net 参数
  optional NetParameter net_param = 25;
  // 网络状态，可有一个训练网络和多个测试网络
  optional NetState train_state = 26;
  repeated NetState test_state = 27;

  // 测试阶段迭代次数
  repeated int32 test_iter = 3;

  // 相邻两个测试阶段的间隔（单位为迭代次数）
  optional int32 test_interval = 4 [default = 0];
  // 测试时是否需要计算损失函数，默认不需要
  optional bool test_compute_loss = 19 [default = false];
  // 初始测试阶段，默认为真，表示在第一次迭代之前首先进行一次测试流程，保证内存可用，打印初始损失函数值
  optional bool test_initialization = 32 [default = true];
  optional float base_lr = 5; // 学习速率，全局通用
  // 每次打印信息的间隔（单位为迭代次数）
  optional int32 display = 6;
  // 显示最近一个迭代周期的损失函数平均值
  optional int32 average_loss = 33 [default = 1];
  optional int32 max_iter = 7; // 最大迭代次数
  // 误差梯度在多少个批量数据上累积，默认为一个
  optional int32 iter_size = 36 [default = 1];

  // 学习速率衰减策略，目前已经实现的有：
  //   - fixed: 固定学习速率，始终等于 base_lr
  //   - step: 步进衰减，等于  $base\_lr * \gamma^{\lfloor iter / step \rfloor}$ 
  //   - exp: 指数衰减，等于  $base\_lr * \gamma^{iter}$ 
  //   - inv: 倒数衰减，等于  $base\_lr * (1 + \gamma * iter)^{-power}$ 
  //   - multistep: 多步衰减，与步进衰减类似，允许非均匀步进值 (stepvalue)
  //   - poly: 多项式衰减，在 max_iter 时达到 0。计算公式为  $base\_lr (1 - iter / max\_iter)^{power}$ 
  //   - sigmoid: sigmoid 衰减，等于  $base\_lr (1 / (1 + \exp(-\gamma * (iter - stepsize))))$ 
```

// 上面出现的 base_lr, max_iter, gamma, step, stepvalue 和 power 都在 solver.prototxt 中定义, iter 是当前迭代次数

```
optional string lr_policy = 8;
optional float gamma = 9;
optional float power = 10;
optional float momentum = 11; // 遗忘因子
optional float weight_decay = 12; // 权值衰减常数
// 规整化类型, L1 或 L2
optional string regularization_type = 29 [default = "L2"];
optional int32 stepsize = 13;
repeated int32 stepvalue = 34;
// 设置为大于 0 的数, 可以保证误差梯度的范数不超过这个边界
optional float clip_gradients = 35 [default = -1];
// 打快照间隔, 单位为迭代次数
optional int32 snapshot = 14 [default = 0];
// 快照文件前缀
optional string snapshot_prefix = 15;
// 打快照时是否包含梯度信息, 默认为 false, 不包含。设置为 true, 有利于调试, 但快照文件会增大
optional bool snapshot_diff = 16 [default = false];
// 快照文件格式, 支持 HDF5 和二进制 ProtoBuffer 两种格式
enum SnapshotFormat {
    HDF5 = 0;
    BINARYPROTO = 1;
}
// 在默认情况下, 快照文件为二进制 ProtoBuffer 格式
optional SnapshotFormat snapshot_format = 37 [default = BINARYPROTO];
// 求解器模式, 有 CPU 和 GPU 两种
enum SolverMode {
    CPU = 0;
    GPU = 1;
}
// 默认为 GPU
optional SolverMode solver_mode = 17 [default = GPU];
// 在 GPU 模式下需指定设备号
optional int32 device_id = 18 [default = 0];
// 随机数种子, 若为非负数, 则每次随机数发生器会以确定方式产生随机数, 利于重现某些结果; 默认使用系统时钟生成随机数种子
optional int64 random_seed = 20 [default = -1];

// 求解器类型
```

```

enum SolverType {
    SGD = 0;
    NESTEROV = 1;
    ADAGRAD = 2;
    RMSPROP = 3;
    ADADELTA = 4;
    ADAM = 5;
}
// 默认为 SGD，随机梯度下降法
optional SolverType solver_type = 30 [default = SGD];
// 在 RMSProp、AdaGrad、AdaDelta 和 Adam 求解器类型情况下保证数值稳定性的参数
optional float delta = 31 [default = 1e-8];
// Adam 求解器参数
optional float momentum2 = 39 [default = 0.999];
// RMS 求解器参数
optional float rms_decay = 38;
// 是否打印调试信息，默认不打印
optional bool debug_info = 23 [default = false];
// 训练结束时是否打快照，默认会打
optional bool snapshot_after_train = 28 [default = true];
}

```

通过这些描述信息，可以大体猜出其实现过程。我们来揭开谜底。

打开 `include/caffe/solver.hpp`，查看 `Solver` 类声明：

```

// 求解器类
template<typename Dtype>
class Solver {
public:
    // 两种显式构造函数，分别从 SolverParameter 对象和 solver 描述文件创建
    explicit Solver(const SolverParameter& param,
        const Solver* root_solver = NULL);
    explicit Solver(const string& param_file, const Solver* root_solver = NULL);
    // 初始化
    void Init(const SolverParameter& param);
    void InitTrainNet(); // 初始化训练 Net
    void InitTestNets(); // 初始化测试 Net
    // 主入口，从一个 resume_file 中恢复训练。如果为 NULL，则从 iter 0 开始训练
    virtual void Solve(const char* resume_file = NULL);
}

```

```

inline void Solve(const string resume_file) { Solve(resume_file.c_str()); }
// 进行第 iter 次迭代
void Step(int iters);
// 从 resume_file 中恢复训练
void Restore(const char* resume_file);
// 虚析构函数
virtual ~Solver() {}
// Getters/Setters
inline const SolverParameter& param() const { return param_; }
inline shared_ptr<Net<Dtype>> net() { return net_; }
inline const vector<shared_ptr<Net<Dtype>>>& test_nets() {
    return test_nets_;
}
int iter() { return iter_; }

protected:
// 对当前迭代产生并应用更新值，纯虚函数，需要到派生类中去查找
virtual void ApplyUpdate() = 0;
// 实现基本打快照工具，存储学习到的网络。应当实现 SnapshotSolverState() 函数，生成 SolverState
// ProtoBuffer，并和学习到的网络一起写入磁盘
void Snapshot();
string SnapshotFilename(const string extension);
string SnapshotToBinaryProto(); // 保存为二进制 ProtoBuffer 文件
string SnapshotToHDF5(); // 保存为 HDF5 文件
virtual void SnapshotSolverState(const string& model_filename) = 0;
virtual void RestoreSolverStateFromHDF5(const string& state_file) = 0;
virtual void RestoreSolverStateFromBinaryProto(const string& state_file) = 0;
// 对网络做测试
void TestAll();
void Test(const int test_net_id = 0);

void DisplayOutputBlobs(const int net_id);

SolverParameter param_; // 用于从 prototxt 中获取参数
int iter_; // 当前迭代次数
int current_step_; // 当前 step 大小，用于学习速率步进衰减策略
shared_ptr<Net<Dtype>> net_; // 若干 Net 对象的指针，用于训练
vector<shared_ptr<Net<Dtype>>> test_nets_; // 若干 Net 对象的指针，用于测试
vector<Callback*> callbacks_; // 回调函数列表

```

```

// 根求解器，用于多机并行训练
const Solver* const root_solver_;

DISABLE_COPY_AND_ASSIGN(Solver);
};

// 用 SGD 方法的求解器类，派生于 Solver
template<typename Dtype>
class SGDSolver : public Solver<Dtype> {
public:
// 显式构造函数，注意在构造时调用了自身的 PreSolve() 函数
explicit SGDSolver(const SolverParameter& param)
    : Solver<Dtype>(param) { PreSolve(); }
explicit SGDSolver(const string& param_file)
    : Solver<Dtype>(param_file) { PreSolve(); }
// 获得权值更新历史值
const vector<shared_ptr<Blob<Dtype>>>& history() { return history_; }

protected:
// 求解前的准备工作
void PreSolve();
// 得到学习速率
Dtype GetLearningRate();
// 应用更新
virtual void ApplyUpdate();
// 对某权值进行归一化
virtual void Normalize(int param_id);
// 对某权值进行规整化
virtual void Regularize(int param_id);
// 计算某权值在特定学习速率下的更新值
virtual void ComputeUpdateValue(int param_id, Dtype rate);
// 梯度抑制
virtual void ClipGradients();
// 与 Solver 中的对应函数功能一致
virtual void SnapshotSolverState(const string& model_filename);
virtual void SnapshotSolverStateToBinaryProto(const string& model_filename);
virtual void SnapshotSolverStateToHDF5(const string& model_filename);
virtual void RestoreSolverStateFromHDF5(const string& state_file);

```

```
virtual void RestoreSolverStateFromBinaryProto(const string& state_file);
// 以下为 SGD 求解时需要的临时存储。history_ 中保留历史增量数据；update_ 中保留更新相关数据，不需要打快照；temp_ 保留在计算梯度/更新值时可能需要的其他信息，不需要打快照
vector<shared_ptr<Blob<Dtype>>> history_, update_, temp_;
DISABLE_COPY_AND_ASSIGN(SGDSolver);
};
```

我们重点研究 SGDSolver 类的实现，其他求解器类如 RMSPropSolver、AdaDeltaSolver、AdamSolver、NesterovSolver、AdaGradSolver 等，请读者结合相关论文做选择性阅读。

首先看 Solver 类的构造函数：

```
template<typename Dtype>
Solver<Dtype>::Solver(const SolverParameter& param, const Solver* root_solver)
    : net_(), callbacks_(), root_solver_(root_solver) {
    Init(param);
}

template<typename Dtype>
Solver<Dtype>::Solver(const string& param_file, const Solver* root_solver)
    : net_(), callbacks_(), root_solver_(root_solver) {
    SolverParameter param;
    ReadProtoFromTextFileOrDie(param_file, &param);
    Init(param);
}
```

调用了 Init() 函数，将 SolverParameter 作为参数传递。进一步看 Init() 函数如下：

```
template<typename Dtype>
void Solver<Dtype>::Init(const SolverParameter& param) {
    param_ = param; // 将外部 SolverParameter 对象拷贝到内部
    Caffe::set_random_seed(param_.random_seed()); // 设置随机数种子
    // 初始化训练网络
    InitTrainNet();
    // 初始化预测网络
    InitTestNets();
    LOG(INFO) << "Solver scaffolding done.";
    // 迭代次数清零
    iter_ = 0;
    // 学习速率步进参数清零
    current_step_ = 0;
}
```

Init()中调用了两个 Net 初始化函数，继续追踪：

```
// 初始化训练网络
template<typename Dtype>
void Solver<Dtype>::InitTrainNet() {
    // 创建 NetParameter 对象
    NetParameter net_param;
    // 从 Solver 中获取 NetParameter 信息
    if (param_.has_train_net_param()) {
        LOG_IF(INFO, Caffe::root_solver())
            << "Creating training net specified in train_net_param.";
        net_param.CopyFrom(param_.train_net_param());
    } else if (param_.has_train_net()) {
        LOG_IF(INFO, Caffe::root_solver())
            << "Creating training net from train_net file: "<< param_.train_net();
        ReadNetParamsFromTextFileOrDie(param_.train_net(), &net_param);
    }
    if (param_.has_net_param()) {
        LOG_IF(INFO, Caffe::root_solver())
            << "Creating training net specified in net_param.";
        net_param.CopyFrom(param_.net_param());
    }
    if (param_.has_net()) {
        LOG_IF(INFO, Caffe::root_solver())
            << "Creating training net from net file: "<< param_.net();
        ReadNetParamsFromTextFileOrDie(param_.net(), &net_param);
    }
    // 设置当前训练网络状态
    NetState net_state;
    net_state.set_phase(TRAIN);
    net_state.MergeFrom(net_param.state());
    net_state.MergeFrom(param_.train_state());
    net_param.mutable_state()->CopyFrom(net_state);
    if (Caffe::root_solver()) {
        // 创建训练网络
        net_.reset(new Net<Dtype>(net_param));
    } else {
        net_.reset(new Net<Dtype>(net_param, root_solver_->net_.get()));
    }
}
```



```

// 初始化预测网络
template<typename Dtype>
void Solver<Dtype>::InitTestNets() {
    // 确定当前 Solver 里面到底有多少个预测网络, 略
    int test_net_id = 0;
    vector<string> sources(num_test_net_instances);
    vector<NetParameter> net_params(num_test_net_instances);
    // 为每个预测网络设置参数, 参数来源可能是 solver.prototxt 中的 test_net_param 条目或
    solver.prototxt 中指定的 test_net 文件
    for (int i = 0; i < num_test_net_params; ++i, ++test_net_id) {
        sources[test_net_id] = "test_net_param";
        net_params[test_net_id].CopyFrom(param_.test_net_param(i));
    }
    for (int i = 0; i < num_test_net_files; ++i, ++test_net_id) {
        sources[test_net_id] = "test_net file: " + param_.test_net(i);
        ReadNetParamsFromTextFileOrDie(param_.test_net(i),
            &net_params[test_net_id]);
    }
    // 剩下的预测网络参数也做初始化
    const int remaining_test_nets = param_.test_iter_size() - test_net_id;
    if (has_net_param) {
        for (int i = 0; i < remaining_test_nets; ++i, ++test_net_id) {
            sources[test_net_id] = "net_param";
            net_params[test_net_id].CopyFrom(param_.net_param());
        }
    }
    if (has_net_file) {
        for (int i = 0; i < remaining_test_nets; ++i, ++test_net_id) {
            sources[test_net_id] = "net file: " + param_.net(i);
            ReadNetParamsFromTextFileOrDie(param_.net(i), &net_params[test_net_id]);
        }
    }
    // 初始化内置变量 test_nets_
    test_nets_.resize(num_test_net_instances);
    for (int i = 0; i < num_test_net_instances; ++i) {
        // 设置每个预测网络状态
        NetState net_state;
        net_state.set_phase(TEST);
        net_state.MergeFrom(net_params[i].state());
    }
}

```

```

    if (param_.test_state_size()) {
        net_state.MergeFrom(param_.test_state(i));
    }
    net_params[i].mutable_state()->CopyFrom(net_state);
    LOG(INFO)
        << "Creating test net (" << i << ") specified by " << sources[i];
    if (Caffe::root_solver()) {
        // 创建每个预测网络
        test_nets_[i].reset(new Net<Dtype>(net_params[i]));
    } else {
        test_nets_[i].reset(new Net<Dtype>(net_params[i],
            root_solver_->test_nets_[i].get()));
    }
    // 设置调试信息
    test_nets_[i]->set_debug_info(param_.debug_info());
}
}

```

从这里清晰地看到，Solver 包含一个训练网络 net_对象和若干个预测网络 test_nets_对象，由 Solver 统一管理它们的运行。结合前两天内容，我们可以想起在创建 Net 对象期间会调用一系列 Layer SetUp()函数，脑海中闪现出一群忙忙碌碌的身影。

13.2.3 CNN 训练过程

我们前面熟悉了 Net Forward/Backward 计算过程，这还不够，需要进一步研究其触发时机，这样才能了解 CNN 整体运作原理。为此，我们深入阅读 Solver 源码的 Solve()部分。

```

// 求解器的核心函数
template<typename Dtype>
void Solver<Dtype>::Solve(const char* resume_file) {
    CHECK(Caffe::root_solver());
    LOG(INFO) << "Solving " << net_->name();
    LOG(INFO) << "Learning Rate Policy: " << param_.lr_policy();

    if (resume_file) {
        // 如果指定了快照恢复文件，则从快照恢复训练环境
        LOG(INFO) << "Restoring previous solver status from " << resume_file;
        Restore(resume_file);
    }
}

```

```

}

int start_iter = iter_;
Step(param_.max_iter() - iter_); // 关键函数
// 求解后打一次快照。可以在 solver.prototxt 中显式说明 snapshot_after_train := false 来禁用
// 这个功能
if (param_.snapshot_after_train()
    && (!param_.snapshot() || iter_ % param_.snapshot() != 0)) {
    Snapshot();
}
// 全部求解完成后，再进行一次额外的训练和预测，显示其损失函数
if (param_.display() && iter_ % param_.display() == 0) {
    Dtype loss;
    net_>ForwardPrefilled(&loss);
    LOG(INFO) << "Iteration " << iter_ << ", loss = " << loss;
}
if (param_.test_interval() && iter_ % param_.test_interval() == 0) {
    TestAll();
}
LOG(INFO) << "Optimization Done.";
}

```

继续深入 Step()这个函数：

```

// 求解器迭代过程
template<typename Dtype>
void Solver<Dtype>::Step(int iters) {
    // 入口参数 iters 表示需要循环这么多次
    const int start_iter = iter_;
    const int stop_iter = iter_ + iters;
    int average_loss = this->param_.average_loss();
    losses_.clear();
    smoothed_loss_ = 0;
    // 循环开始
    while (iter_ < stop_iter) {
        // 清零训练网络的所有权值 diff
        net_>ClearParamDiffs();
        if (param_.test_interval() && iter_ % param_.test_interval() == 0
            && (iter_ > 0 || param_.test_initialization())
            && Caffe::root_solver()) {

```

```

// 周期性预测，评估网络质量
TestAll();
if (requested_early_exit_) {
    // Break out of the while loop because stop was requested while testing.
    break;
}
}

const bool display = param_.display() && iter_ % param_.display() == 0;
net_>set_debug_info(display && param_.debug_info());
// 损失函数、误差值累加
Dtype loss = 0;
for (int i = 0; i < param_.iter_size(); ++i) {
    loss += net_>ForwardBackward();
}
// 取平均
loss /= param_.iter_size();

// 平滑滤波
UpdateSmoothedLoss(loss, start_iter, average_loss);
if (display) {
    // 打印当前（平滑后的）损失函数值
    LOG_IF(INFO, Caffe::root_solver()) << "Iteration " << iter_
        << ", loss = " << smoothed_loss_;
    // 获取训练网络输出 Blob，格式化之后打印输出
    const vector<Blob<Dtype>*>& result = net_>output_blobs();
    int score_index = 0;
    for (int j = 0; j < result.size(); ++j) {
        const Dtype* result_vec = result[j]>cpu_data();
        const string& output_name =
            net_>blob_names()[net_>output_blob_indices()[j]];
        const Dtype loss_weight =
            net_>blob_loss_weights()[net_>output_blob_indices()[j]];
        for (int k = 0; k < result[j]>count(); ++k) {
            ostringstream loss_msg_stream;
            if (loss_weight) {
                loss_msg_stream << " (* " << loss_weight
                    << " = " << loss_weight * result_vec[k] << " loss)";
            }
        }
    }
}

```

```

    LOG_IF(INFO, Caffe::root_solver()) << "    Train net output #"
        << score_index++ << ": " << output_name << " = "
        << result_vec[k] << loss_msg_stream.str();
    }
}
}
// 应用更新
ApplyUpdate();
// 在 Solver 类中这是个纯虚函数，实现要到派生类如 SGDSolver 中查看

// 迭代次数递增
++iter_;
// 如果需要，打个快照
if ((param_.snapshot()
    && iter_ % param_.snapshot() == 0
    && Caffe::root_solver()) ||
    (request == SolverAction::SNAPSHOT)) {
    Snapshot();
}
}
}

```

由于 Step()函数调用了纯虚函数 ApplyUpdate()，我们需要将注意力转向 SGDSolver 中的对应函数查看实现过程。

```

// 应用更新
template<typename Dtype>
void SGDSolver<Dtype>::ApplyUpdate() {
    // 首先，获取学习速率，必要时打印
    Dtype rate = GetLearningRate();
    if (this->param_.display() && this->iter_ % this->param_.display() == 0) {
        LOG(INFO) << "Iteration " << this->iter_ << ", lr = " << rate;
    }
    // 钳制误差梯度
    ClipGradients();
    // 对训练网络中每个权值 Blob，都做归一化、正则化、计算增量这三步
    for (int param_id = 0; param_id < this->net_->learnable_params().size();
        ++param_id) {
        Normalize(param_id);
        Regularize(param_id);
    }
}

```

```

    ComputeUpdateValue(param_id, rate);
}
// 调用训练网络的 Update() 函数，实现见第 12 天内容。
this->net_->Update();
}

// 返回当前学习速率，依据所选的学习策略和当前迭代进度进行计算
template<typename Dtype>
Dtype SGDSolver<Dtype>::GetLearningRate() {
    Dtype rate;
    const string& lr_policy = this->param_.lr_policy();
    if (lr_policy == "fixed") {
        rate = this->param_.base_lr();
    } else if (lr_policy == "step") {
        this->current_step_ = this->iter_ / this->param_.stepsize();
        rate = this->param_.base_lr() *
            pow(this->param_.gamma(), this->current_step_);
    } else if (lr_policy == "exp") {
        rate = this->param_.base_lr() * pow(this->param_.gamma(), this->iter_);
    } else if (lr_policy == "inv") {
        rate = this->param_.base_lr() *
            pow(Dtype(1) + this->param_.gamma() * this->iter_,
                - this->param_.power());
    } else if (lr_policy == "multistep") {
        if (this->current_step_ < this->param_.stepvalue_size() &&
            this->iter_ >= this->param_.stepvalue(this->current_step_)) {
            this->current_step_++;
            LOG(INFO) << "MultiStep Status: Iteration " <<
                this->iter_ << ", step = " << this->current_step_;
        }
        rate = this->param_.base_lr() *
            pow(this->param_.gamma(), this->current_step_);
    } else if (lr_policy == "poly") {
        rate = this->param_.base_lr() * pow(Dtype(1.) -
            (Dtype(this->iter_) / Dtype(this->param_.max_iter()))),
            this->param_.power());
    } else if (lr_policy == "sigmoid") {
        rate = this->param_.base_lr() * (Dtype(1.) /
            (Dtype(1.) + exp(-this->param_.gamma() * (Dtype(this->iter_) -

```

```

        Dtype(this->param_.stepsize()))));
    } else {
        LOG(FATAL) << "Unknown learning rate policy: " << lr_policy;
    }
    return rate;
}

// 预求解, 在 SGDSolver 构造函数中调用
template<typename Dtype>
void SGDSolver<Dtype>::PreSolve() {
    // 初始化临时变量, 尺寸与训练网络的权值完全一致
    const vector<Blob<Dtype>*>& net_params = this->net_->learnable_params();
    history_.clear();
    update_.clear();
    temp_.clear();
    for (int i = 0; i < net_params.size(); ++i) {
        const vector<int>& shape = net_params[i]->shape();
        history_.push_back(shared_ptr<Blob<Dtype>>(new Blob<Dtype>(shape)));
        update_.push_back(shared_ptr<Blob<Dtype>>(new Blob<Dtype>(shape)));
        temp_.push_back(shared_ptr<Blob<Dtype>>(new Blob<Dtype>(shape)));
    }
}

// 沿 iter 维度归一化, 即 diff 全都除以 iter_size
template<typename Dtype>
void SGDSolver<Dtype>::Normalize(int param_id) {
    if (this->param_.iter_size() == 1) { return; }
    const vector<Blob<Dtype>*>& net_params = this->net_->learnable_params();
    const Dtype accum_normalization = Dtype(1.) / this->param_.iter_size();
    switch (Caffe::mode()) {
    case Caffe::CPU: {
        caffe_scal(net_params[param_id]->count(), accum_normalization,
            net_params[param_id]->mutable_cpu_diff());
        break;
    }
    case Caffe::GPU: {
        break;
    }
    default:
        LOG(FATAL) << "Unknown caffe mode: " << Caffe::mode();

```

```

    }
}
// 正则化
template<typename Dtype>
void SGDSolver<Dtype>::Regularize(int param_id) {
    const vector<Blob<Dtype>*>& net_params = this->net_->learnable_params();
    const vector<float>& net_params_weight_decay =
        this->net_->params_weight_decay();
    // 拿到 solver.prototxt 中的 weight_decay
    Dtype weight_decay = this->param_.weight_decay();
    // 拿到 solver.prototxt 中的 regularization_type
    string regularization_type = this->param_.regularization_type();
    // 获得每个层自己本地的 weight_decay = solver_weight_decay * 本地缩放因子
    Dtype local_decay = weight_decay * net_params_weight_decay[param_id];
    switch (Caffe::mode()) {
    case Caffe::CPU: {
        if (local_decay) {
            if (regularization_type == "L2") {
                // L2 正则化, 引入 decay * W 项
                caffe_axpy(net_params[param_id]->count(),
                    local_decay,
                    net_params[param_id]->cpu_data(),
                    net_params[param_id]->mutable_cpu_diff());
            } else if (regularization_type == "L1") {
                // L1 正则化, 引入 decay * sign(W) 项
                caffe_cpu_sign(net_params[param_id]->count(),
                    net_params[param_id]->cpu_data(),
                    temp_[param_id]->mutable_cpu_data()); // 符号函数
                caffe_axpy(net_params[param_id]->count(),
                    local_decay,
                    temp_[param_id]->cpu_data(),
                    net_params[param_id]->mutable_cpu_diff());
            } else {
                LOG(FATAL) << "Unknown regularization type: " << regularization_type;
            }
        }
        break;
    }
    case Caffe::GPU: {

```



```

    break;
}
default:
    LOG(FATAL) << "Unknown caffe mode: " << Caffe::mode();
}
}
// 计算权值增量
template<typename Dtype>
void SGDSolver<Dtype>::ComputeUpdateValue(int param_id, Dtype rate) {
    const vector<Blob<Dtype>*>& net_params = this->net_->learnable_params();
    const vector<float>& net_params_lr = this->net_->params_lr();
    Dtype momentum = this->param_.momentum();
    // 获得当前层的学习速率, 数值上等于全局学习速率乘以本地缩放因子
    Dtype local_rate = rate * net_params_lr[param_id];
    // 计算增量存放到 history_, 之后拷贝到权值的 diff 域
    switch (Caffe::mode()) {
    case Caffe::CPU: {
        // 学习速率乘以增量 + 遗忘因子乘以旧的增量 = 新的增量
        caffe_cpu_axpby(net_params[param_id]->count(), local_rate,
            net_params[param_id]->cpu_diff(), momentum,
            history_[param_id]->mutable_cpu_data());
        caffe_copy(net_params[param_id]->count(),
            history_[param_id]->cpu_data(),
            net_params[param_id]->mutable_cpu_diff());
        break;
    }
    case Caffe::GPU: {
        break;
    }
    default:
        LOG(FATAL) << "Unknown caffe mode: " << Caffe::mode();
    }
}
}

```

13.2.4 CNN 预测过程

Solver 每隔一定周期会对训练的网络做一次评估, 使用 `test_nets_` 切换到新数据集进行预测。预测时调用 `TestAll()` 函数:

```

template<typename Dtype>
void Solver<Dtype>::TestAll() {
    // 遍历 test_nets_ 中的每个对象
    for (int test_net_id = 0; test_net_id < test_nets_.size(); ++test_net_id) {
        Test(test_net_id);
    }
}

// 对单个 Net 进行评估
template<typename Dtype>
void Solver<Dtype>::Test(const int test_net_id) {
    CHECK(Caffe::root_solver());
    LOG(INFO) << "Iteration " << iter_
        << ", Testing net (" << test_net_id << ")";
    CHECK_NOTNULL(test_nets_[test_net_id].get())->
        ShareTrainedLayersWith(net_.get());
    vector<Dtype> test_score;
    vector<int> test_score_output_id;
    vector<Blob<Dtype>*> bottom_vec;
    // 获得单个 test_net 对象
    const shared_ptr<Net<Dtype>>& test_net = test_nets_[test_net_id];
    Dtype loss = 0;
    // 迭代次数在 solver.prototxt 中由 test_iter 设定
    for (int i = 0; i < param_.test_iter(test_net_id); ++i) {
        Dtype iter_loss;
        // 预测网络执行前向传播计算
        const vector<Blob<Dtype>*>& result =
            test_net->Forward(bottom_vec, &iter_loss);
        if (param_.test_compute_loss()) {
            // 记录 loss 值
            loss += iter_loss;
        }
        if (i == 0) {
            for (int j = 0; j < result.size(); ++j) {
                const Dtype* result_vec = result[j]->cpu_data();
                for (int k = 0; k < result[j]->count(); ++k) {
                    test_score.push_back(result_vec[k]);
                    test_score_output_id.push_back(j);
                }
            }
        }
    }
}

```

```

} else {
    int idx = 0;
    for (int j = 0; j < result.size(); ++j) {
        const Dtype* result_vec = result[j]->cpu_data();
        for (int k = 0; k < result[j]->count(); ++k) {
            test_score[idx++] += result_vec[k];
        }
    }
}

// 打印损失函数值
if (param_.test_compute_loss()) {
    loss /= param_.test_iter(test_net_id);
    LOG(INFO) << "Test loss: " << loss;
}

// 打印准确率、损失函数
for (int i = 0; i < test_score.size(); ++i) {
    const int output_blob_index =
        test_net->output_blob_indices()[test_score_output_id[i]];
    const string& output_name = test_net->blob_names()[output_blob_index];
    const Dtype loss_weight = test_net->blob_loss_weights()[output_blob_index];
    ostringstream loss_msg_stream;
    const Dtype mean_score = test_score[i] / param_.test_iter(test_net_id);
    if (loss_weight) {
        loss_msg_stream << " (* " << loss_weight
            << " = " << loss_weight * mean_score << " loss)";
    }
    LOG(INFO) << "    Test net output #" << i << ": " << output_name << " = "
        << mean_score << loss_msg_stream.str();
}
}

```

13.2.5 Solver 的快照和恢复功能

求解器会在 `Solver::Snapshot()` 和 `Solver::SnapshotSolverState()` 中将权值和它训练时的状态打快照。权值快照将学习到的模型导出，而求解器快照允许从特定快照点恢复训练。使用 `Solver::Restore()` 和 `Solver::RestoreSolverState()` 恢复训练。权值保存到 *.caffemodel 文件中，求解器状态保存到 *.solverstate 文件中。每个文件都有 _iter_N 后缀，用于标记打快照时的训练迭代

次数。

首先我们看一下在 `caffe.proto` 中对求解器快照数据结构的描述：

```
// 求解器打快照时会用到
message SolverState {
    optional int32 iter = 1; // 当前迭代次数
    optional string learned_net = 2; // 保存权值的文件 (*.caffemodel)
    repeated BlobProto history = 3; // SGD 求解器的历史数据
    optional int32 current_step = 4 [default = 0]; // 当前学习速率的 step 值
}
```

在求解器描述 `prototxt` 文件中，可以对快照功能做如下配置：

```
snapshot: 5000           // 快照间隔，每 5000 次迭代打一次快照
snapshot_prefix: "/path/to/model" // 快照文件前缀，对 caffemodel 和 solverstate 文件均有效。注意该路径是相对于 ./build/tools/caffe，而不是 solver.prototxt
snapshot_diff: false // 打快照时是否包括 diff，有利于调试。但会增加文件大小
snapshot_after_train: true // 在训练结束时，是否追加一个快照
```

我们具体看一下代码实现：

```
// Solver 的快照功能
template<typename Dtype>
void Solver<Dtype>::Snapshot() {
    CHECK(Caffe::root_solver());
    string model_filename;
    // 权值快照支持两种格式：HDF5 和二进制 ProtoBuffer，根据 solver.prototxt 中的 snapshot_format 设置而定
    switch (param_.snapshot_format()) {
        case caffe::SolverParameter_SnapshotFormat_BINARYPROTO:
            model_filename = SnapshotToBinaryProto();
            break;
        case caffe::SolverParameter_SnapshotFormat_HDF5:
            model_filename = SnapshotToHDF5();
            break;
        default:
            LOG(FATAL) << "Unsupported snapshot format.";
    }
    // 求解器打快照
    SnapshotSolverState(model_filename);
}
```

```

// 获得快照全名
template<typename Dtype>
string Solver<Dtype>::SnapshotFilename(const string extension) {
    // 前缀由 solver.prototxt 中的 snapshot_prefix 指定
    string filename(param_.snapshot_prefix());
    const int kBufferSize = 20;
    char iter_str_buffer[kBufferSize];
    snprintf(iter_str_buffer, kBufferSize, "_iter_%d", iter_);
    // 名称加入迭代次数, 方便区分
    // 后缀为 ".solverstate" 或 ".caffemodel"
    return filename + iter_str_buffer + extension;
}

// 为权值打快照, 格式为二进制 ProtoBuffer
template<typename Dtype>
string Solver<Dtype>::SnapshotToBinaryProto() {
    string model_filename = SnapshotFilename(".caffemodel");
    LOG(INFO) << "Snapshotting to binary proto file " << model_filename;
    // 生成快照名
    NetParameter net_param;
    // 将训练网络从内存导出到 ProtoBuffer 对象
    net_ -> ToProto(&net_param, param_.snapshot_diff());
    // 再从 ProtoBuffer 对象写入磁盘文件
    WriteProtoToBinaryFile(net_param, model_filename);
    return model_filename;
}

// 为权值打快照, 格式为 HDF5
template<typename Dtype>
string Solver<Dtype>::SnapshotToHDF5() {
    // 生成快照名
    string model_filename = SnapshotFilename(".caffemodel.h5");
    LOG(INFO) << "Snapshotting to HDF5 file " << model_filename;
    // 直接调用 Net 自带的 ToHDF5() 函数, 写入 HDF5 文件
    net_ -> ToHDF5(model_filename, param_.snapshot_diff());
    return model_filename;
}

// 从快照恢复功能
template<typename Dtype>
void Solver<Dtype>::Restore(const char* state_file) {
    CHECK(Caffe::root_solver());

```

```
string state_filename(state_file);  
// 根据有无 .h5 后缀来判断快照格式，从而选择相应的恢复方法  
if (state_filename.size() >= 3 &&  
    state_filename.compare(state_filename.size() - 3, 3, ".h5") == 0) {  
    RestoreSolverStateFromHDF5(state_filename);  
} else {  
    RestoreSolverStateFromBinaryProto(state_filename);  
}  
}
```

通过这三天介绍的内容，我们学到了深度学习最核心的几项技术：多表示层、激活函数、损失函数、反向传播、最大梯度下降法优化求解，这些技术充满了挑战和机遇，需要较深的理论功底和一定的编程技巧，值得读者细细揣摩。通过对深度学习最新理论的不断学习和编程实践，将会让你成长为优秀的算法工程师。

13.3 练习题

1. 尝试使用不同的学习策略运行 MNIST、CIFAR10 训练例程，观察损失函数和准确率变化情况。
2. 尝试使用不同的求解器训练网络。

第 14 天

Caffe 实用工具

Caffe 框架编译之后会生成动态链接库 `libcaffe.so`，其本身并不能独立运行。如果需要运行 Caffe，则需要写一个 `main()` 函数，调用 Caffe 的 API，编译时包含相应的头文件，链接时加入 `libcaffe.so`，这样才能构成一个完整的 Caffe 应用程序。在 `tools/` 目录下的就是一些调用 `libcaffe.so` 的实用工具源码。我们在第 2 天跑 Caffe 例程时使用的 `build/tools/caffe.bin`，也是其中一个实用工具。

前面用庖丁解牛的方法研究了 Caffe 各个组成部分的运行机理，而今天内容则关注功能性，即如何利用 Caffe 强大的模块构建符合应用需求的完整工程。

14.1 训练和预测

通过 Caffe 例程我们已经了解到，只需通过命令行向 `caffe.bin` 传递不同参数（`train/test`）就可以实现深度神经网络的训练、预测。下面研究其源码 `tools/caffe.cpp` 的具体实现。

```
#ifndef WITH_PYTHON_LAYER
#include "boost/python.hpp"
namespace bp = boost::python;
#endif

#include <glog/logging.h>    // 包含这个文件，就可以使用 GLOG

#include <cstring>
#include <map>
#include <string>
#include <vector>
```

```

#include "boost/algorithm/string.hpp" // 字符串处理库
#include "caffe/caffe.hpp" // 只需包含这个头文件，即可使用 Caffe 的所有组件 (Blob, Layer, Net,
Solver, ...)

using caffe::Blob;
using caffe::Caffe;
using caffe::Net;
using caffe::Layer;
using caffe::Solver;
using caffe::shared_ptr;
using caffe::string;
using caffe::Timer;
using caffe::vector;
using std::ostringstream; // 借此机会学习 C++命名空间用法

DEFINE_string(gpu, "",
    "Optional; run in GPU mode on given device IDs separated by ','."
    "Use '-gpu all' to run on all available GPUs. The effective training "
    "batch size is multiplied by the number of devices.");
DEFINE_string(solver, "",
    "The solver definition protocol buffer text file.");
DEFINE_string(model, "",
    "The model definition protocol buffer text file..");
DEFINE_string(snapshot, "",
    "Optional; the snapshot solver state to resume training.");
DEFINE_string(weights, "",
    "Optional; the pretrained weights to initialize finetuning, "
    "separated by ','. Cannot be set simultaneously with snapshot.");
DEFINE_int32(iterations, 50,
    "The number of iterations to run.");

// 简单的 caffe 命令注册表
typedef int (*BrewFunction)();
typedef std::map<caffe::string, BrewFunction> BrewMap;
BrewMap g_brew_map;

#define RegisterBrewFunction(func) \
namespace { \
class __Registerer_##func { \

```



```

public: /* NOLINT */ \
    __Registerer_##func() { \
        g_brew_map[#func] = &func; \
    } \
}; \
__Registerer_##func g_registerer_##func; \
}

static BrewFunction GetBrewFunction(const caffe::string& name) {
    if (g_brew_map.count(name)) {
        return g_brew_map[name];
    } else {
        LOG(ERROR) << "Available caffe actions:";
        for (BrewMap::iterator it = g_brew_map.begin();
            it != g_brew_map.end(); ++it) {
            LOG(ERROR) << "\t" << it->first;
        }
        LOG(FATAL) << "Unknown action: " << name;
        return NULL; // 该语句不可达，只是为了减少旧编译器警告
    }
}

// 解析 GPU id, 或使用所有可用的 GPU 设备
static void get_gpus(vector<int>* gpus) {
    if (FLAGS_gpu == "all") {
        int count = 0;
#ifdef CPU_ONLY
        CUDA_CHECK(cudaGetDeviceCount(&count));
#else
        NO_GPU;
#endif
        for (int i = 0; i < count; ++i) {
            gpus->push_back(i);
        }
    } else if (FLAGS_gpu.size()) {
        vector<string> strings;
        boost::split(strings, FLAGS_gpu, boost::is_any_of(","));
        for (int i = 0; i < strings.size(); ++i) {
            gpus->push_back(boost::lexical_cast<int>(strings[i]));
        }
    }
}

```

```

    }
} else {
    CHECK_EQ(gpus->size(), 0);
}
}

// caffe 命令, 格式为:
//   caffe <command><args>
//
// 如果需要增加一个命令, 定义一个函数 int command(), 然后这样注册:
// RegisterBrewFunction(action);

// 设备查询命令, 显示 GPU 设备诊断信息
int device_query() {
    LOG(INFO) << "Querying GPUs " << FLAGS_gpu;
    vector<int> gpus;
    get_gpus(&gpus);
    for (int i = 0; i < gpus.size(); ++i) {
        caffe::Caffe::SetDevice(gpus[i]);
        caffe::Caffe::DeviceQuery();
    }
    return 0;
}
RegisterBrewFunction(device_query);

// 从指定的 caffemodel 中向训练、预测网络载入训练过的权值
void CopyLayers(caffe::Solver<float>* solver, const std::string& model_list) {
    std::vector<std::string> model_names;
    boost::split(model_names, model_list, boost::is_any_of(",") );
    for (int i = 0; i < model_names.size(); ++i) {
        LOG(INFO) << "Finetuning from " << model_names[i];
        solver->net()->CopyTrainedLayersFrom(model_names[i]);
        for (int j = 0; j < solver->test_nets().size(); ++j) {
            solver->test_nets()[j]->CopyTrainedLayersFrom(model_names[i]);
        }
    }
}
}

```

```

// 训练/精调一个模型
int train() {
    CHECK_GT(FLAGS_solver.size(), 0) << "Need a solver definition to train.";
    CHECK(!FLAGS_snapshot.size() || !FLAGS_weights.size())
        << "Give a snapshot to resume training or weights to finetune "
        "but not both.";

    caffe::SolverParameter solver_param;
    caffe::ReadProtoFromTextFileOrDie(FLAGS_solver, &solver_param);

    // 如果未提供 gpus 标志, 则从 solver prototxt 获得计算模式和设备
    if (FLAGS_gpu.size() == 0)
        && solver_param.solver_mode() == caffe::SolverParameter_SolverMode_GPU) {
        if (solver_param.has_device_id()) {
            FLAGS_gpu = "" +
                boost::lexical_cast<string>(solver_param.device_id());
        } else { // 设置默认 GPU 设备号
            FLAGS_gpu = "" + boost::lexical_cast<string>(0);
        }
    }

    vector<int> gpus;
    get_gpus(&gpus);
    if (gpus.size() == 0) {
        Caffe::set_mode(Caffe::CPU);
    } else {
        ostringstream s;
        for (int i = 0; i < gpus.size(); ++i) {
            s << (i ? ", " : "") << gpus[i];
        }
        LOG(INFO) << "Using GPUs " << s.str();

        solver_param.set_device_id(gpus[0]);
        Caffe::SetDevice(gpus[0]);
        Caffe::set_mode(Caffe::GPU);
        Caffe::set_solver_count(gpus.size());
    }
}

```

```

shared_ptr<Solver<float>>solver(caffe::GetSolver<float>(solver_param));

if (FLAGS_snapshot.size()) {
    LOG(INFO) << "Resuming from " << FLAGS_snapshot;
    solver->Restore(FLAGS_snapshot.c_str());
} else if (FLAGS_weights.size()) {
    CopyLayers(solver.get(), FLAGS_weights);
}

if (gpus.size() > 1) {
    caffe::P2PSync<float>sync(solver, NULL, solver->param());
    sync.run(gpus);
} else {
    LOG(INFO) << "Starting Optimization";
    solver->Solve();
}
LOG(INFO) << "Optimization Done.";
return 0;
}

RegisterBrewFunction(train);

// 预测：使用模型打分
int test() {
    CHECK_GT(FLAGS_model.size(), 0) << "Need a model definition to score.";
    CHECK_GT(FLAGS_weights.size(), 0) << "Need model weights to score.";

    // 设置设备 id 和模式
    vector<int> gpus;
    get_gpus(&gpus);
    if (gpus.size() != 0) {
        LOG(INFO) << "Use GPU with device ID " << gpus[0];
        Caffe::SetDevice(gpus[0]);
        Caffe::set_mode(Caffe::GPU);
    } else {
        LOG(INFO) << "Use CPU.";
        Caffe::set_mode(Caffe::CPU);
    }

    // 实例化 caffe net 对象

```

```

Net<float> caffe_net(FLAGS_model, caffe::TEST);
caffe_net.CopyTrainedLayersFrom(FLAGS_weights);
LOG(INFO) << "Running for " << FLAGS_iterations << " iterations.";

vector<Blob<float>*> > bottom_vec;
vector<int> test_score_output_id;
vector<float> test_score;
float loss = 0;
for (int i = 0; i < FLAGS_iterations; ++i) {
    float iter_loss;
    const vector<Blob<float>*>& result =
        caffe_net.Forward(bottom_vec, &iter_loss);
    loss += iter_loss;
    int idx = 0;
    for (int j = 0; j < result.size(); ++j) {
        const float* result_vec = result[j]->cpu_data();
        for (int k = 0; k < result[j]->count(); ++k, ++idx) {
            const float score = result_vec[k];
            if (i == 0) {
                test_score.push_back(score);
                test_score_output_id.push_back(j);
            } else {
                test_score[idx] += score;
            }
        }
        const std::string& output_name = caffe_net.blob_names()[
            caffe_net.output_blob_indices()[j]];
        LOG(INFO) << "Batch " << i << ", " << output_name << " = " << score;
    }
}

loss /= FLAGS_iterations;
LOG(INFO) << "Loss: " << loss;
for (int i = 0; i < test_score.size(); ++i) {
    const std::string& output_name = caffe_net.blob_names()[
        caffe_net.output_blob_indices()[test_score_output_id[i]]];
    const float loss_weight = caffe_net.blob_loss_weights()[
        caffe_net.output_blob_indices()[test_score_output_id[i]]];
    std::ostringstream loss_msg_stream;
    const float mean_score = test_score[i] / FLAGS_iterations;

```

```

    if (loss_weight) {
        loss_msg_stream << " (* " << loss_weight
                        << " = " << loss_weight * mean_score << " loss)";
    }
    LOG(INFO) << output_name << " = " << mean_score << loss_msg_stream.str();
}

return 0;
}
RegisterBrewFunction(test);

// 计时：评测模型执行时间
int time() {
    CHECK_GT(FLAGS_model.size(), 0) << "Need a model definition to time.";

    // 设置设备 id 和模式
    vector<int> gpus;
    get_gpus(&gpus);
    if (gpus.size() != 0) {
        LOG(INFO) << "Use GPU with device ID " << gpus[0];
        Caffe::SetDevice(gpus[0]);
        Caffe::set_mode(Caffe::GPU);
    } else {
        LOG(INFO) << "Use CPU.";
        Caffe::set_mode(Caffe::CPU);
    }

    // 实例化一个 caffe net
    Net<float> caffe_net(FLAGS_model, caffe::TRAIN);

    // 做一次干净的前向、反向流程，保证完成存储区分配
    LOG(INFO) << "Performing Forward";
    // 速度测试，网络不需要任何输入 Blob
    float initial_loss;
    caffe_net.Forward(vector<Blob<float>*>(), &initial_loss);
    LOG(INFO) << "Initial loss: " << initial_loss;
    LOG(INFO) << "Performing Backward";
    caffe_net.Backward();
}

```

```

const vector<shared_ptr<Layer<float>>>& layers = caffe_net.layers();
const vector<vector<Blob<float>*>>& bottom_vecs = caffe_net.bottom_vecs();
const vector<vector<Blob<float>*>>& top_vecs = caffe_net.top_vecs();
const vector<vector<bool>>& bottom_need_backward =
    caffe_net.bottom_need_backward();
LOG(INFO) << "*** Benchmark begins ***";
LOG(INFO) << "Testing for " << FLAGS_iterations << " iterations.";
Timer total_timer;
total_timer.Start();
Timer forward_timer;
Timer backward_timer;
Timer timer;
std::vector<double> forward_time_per_layer(layers.size(), 0.0);
std::vector<double> backward_time_per_layer(layers.size(), 0.0);
double forward_time = 0.0;
double backward_time = 0.0;
for (int j = 0; j < FLAGS_iterations; ++j) {
    Timer iter_timer;
    iter_timer.Start();
    forward_timer.Start();
    for (int i = 0; i < layers.size(); ++i) {
        timer.Start();
        layers[i]->Forward(bottom_vecs[i], top_vecs[i]);
        forward_time_per_layer[i] += timer.MicroSeconds();
    }
    forward_time += forward_timer.MicroSeconds();
    backward_timer.Start();
    for (int i = layers.size() - 1; i >= 0; --i) {
        timer.Start();
        layers[i]->Backward(top_vecs[i], bottom_need_backward[i],
                           bottom_vecs[i]);
        backward_time_per_layer[i] += timer.MicroSeconds();
    }
    backward_time += backward_timer.MicroSeconds();
    LOG(INFO) << "Iteration: " << j + 1 << " forward-backward time: "
        << iter_timer.MilliSeconds() << " ms.";
}

```

```

LOG(INFO) << "Average time per layer: ";
for (int i = 0; i < layers.size(); ++i) {
    const caffe::string& layername = layers[i]->layer_param().name();
    LOG(INFO) << std::setfill(' ') << std::setw(10) << layername <<
        "\tforward: " << forward_time_per_layer[i] / 1000 /
        FLAGS_iterations << " ms.";
    LOG(INFO) << std::setfill(' ') << std::setw(10) << layername <<
        "\tbackward: " << backward_time_per_layer[i] / 1000 /
        FLAGS_iterations << " ms.";
}
total_timer.Stop();
LOG(INFO) << "Average Forward pass: " << forward_time / 1000 /
    FLAGS_iterations << " ms.";
LOG(INFO) << "Average Backward pass: " << backward_time / 1000 /
    FLAGS_iterations << " ms.";
LOG(INFO) << "Average Forward-Backward: " << total_timer.MilliSeconds() /
    FLAGS_iterations << " ms.";
LOG(INFO) << "Total Time: " << total_timer.MilliSeconds() << " ms.";
LOG(INFO) << "**** Benchmark ends ****";
return 0;
}

RegisterBrewFunction(time);

int main(int argc, char** argv) {
    // 打印输出到 stderr
    FLAGS_alsologtostderr = 1;
    // 用法信息
    gflags::SetUsageMessage("command line brew\n"
        "usage: caffe <command><args>\n\n"
        "commands:\n"
        "  train          train or finetune a model\n"
        "  test           score a model\n"
        "  device_query   show GPU diagnostic information\n"
        "  time          benchmark model execution time");
    // 运行工具或显示使用信息
    caffe::GlobalInit(&argc, &argv);
    if (argc == 2) {
#ifdef WITH_PYTHON_LAYER
        try {

```



```

#endif
    return GetBrewFunction(caffe::string(argv[1]))();
#ifdef WITH_PYTHON_LAYER
    } catch (bp::error_already_set) {
        PyErr_Print();
        return 1;
    }
#endif
    } else {
        gflags::ShowUsageWithFlagsRestrict(argv[0], "tools/caffe");
    }
}

```

有关 Caffe 训练选项、Caffe 预测选项、多 GPU、finetune（精调）的内容都在这个主程序里面。

14.2 特征提取

在一些在线应用中，Caffe 常用作图像特征提取器。使用特征提取器可以有效降低图像数据维度，从而降低传输带宽。对特征进一步精细分类可以使用其他分类器（如 SVM）实现。

Caffe 提供的实用工具 `build/tools/extract_features.bin` 实现了特征提取功能，该程序需要一个训练好的网络和一个数据输入层，运行后可得到相应数据通过网络某个中间层产生的特征图并保存到磁盘。该工具用法如下：

```

$ extract_features \ // 可执行程序
pretrained_net_param \ // 预训练的网络，*.caffemodel
feature_extraction_proto_file \ // 网络描述文件，*.prototxt
extract_feature_blob_name1[,name2,...] \ // 需要提取的 Blob 名
save_feature_dataset_name1[,name2,...] \ // 保存特征名
num_mini_batches \ // 做特征提取的数据批量数目
db_type \ // 输入数据的格式，LMDB 或 LEVELDB
[CPU/GPU] \ // 使用 CPU 还是 GPU
[DEVICE_ID=0] // 如果使用 GPU，则选择设备编号

```

例子：

```

$ ./build/tools/extract_features.bin \
models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel \

```

```
models/bvlc_reference_caffenet/train_val.prototxt \
fc6,fc7,fc8 \
myfc6,myfc7,myfc8 \
10 \
lmdb\
GPU \
1
```

```
E1107 05:47:51.462666 21761 extract_features.cpp:54] Using GPU
E1107 05:47:51.463057 21761 extract_features.cpp:60] Using Device_id=1
E1107 05:47:58.369420 21761 extract_features.cpp:135] Extracting Features
E1107 05:48:00.384551 21761 extract_features.cpp:181] Extracted features of 500 query
images for feature blob fc6
E1107 05:48:00.662539 21761 extract_features.cpp:181] Extracted features of 500 query
images for feature blob fc7
E1107 05:48:00.875463 21761 extract_features.cpp:181] Extracted features of 500 query
images for feature blob fc8
E1107 05:48:01.086578 21761 extract_features.cpp:186] Successfully extracted the
features!
$ ls -l myfc*
myfc6:
total 12032
-rw-rw-r-- 1 yourname yourname 12316672 Nov  7 05:48 data.mdb
-rw-rw-r-- 1 yourname yourname    8192 Nov  7 05:47 lock.mdb

myfc7:
total 12032
-rw-rw-r-- 1 yourname yourname 12316672 Nov  7 05:48 data.mdb
-rw-rw-r-- 1 yourname yourname    8192 Nov  7 05:47 lock.mdb

myfc8:
total 4032
-rw-rw-r-- 1 yourname yourname 4124672 Nov  7 05:48 data.mdb
-rw-rw-r-- 1 yourname yourname    8192 Nov  7 05:47 lock.mdb
```

下面看代码实现 tools/extract_features.cpp:

```
#include <stdio.h> // for snprintf
#include <string>
#include <vector>
```

```

#include "boost/algorithm/string.hpp"
#include "google/protobuf/text_format.h"

#include "caffe/blob.hpp"
#include "caffe/common.hpp"
#include "caffe/net.hpp"
#include "caffe/proto/caffe.pb.h"
#include "caffe/util/db.hpp"
#include "caffe/util/io.hpp"
#include "caffe/vision_layers.hpp"

using caffe::Blob;
using caffe::Caffe;
using caffe::Datum;
using caffe::Net;
using boost::shared_ptr;
using std::string;
namespace db = caffe::db;
// 提前声明处理函数
template<typename Dtype>
int feature_extraction_pipeline(int argc, char** argv);

int main(int argc, char** argv) {
    // 控制权交给处理函数
    return feature_extraction_pipeline<float>(argc, argv);
    // return feature_extraction_pipeline<double>(argc, argv);
}

// 处理函数实现
template<typename Dtype>
int feature_extraction_pipeline(int argc, char** argv) {
    ::google::InitGoogleLogging(argv[0]); // 初始化日志系统
    const int num_required_args = 7;
    if (argc < num_required_args) { // 判断命令行参数个数, 小于7个则报错
        LOG(ERROR) <<
            "This program takes in a trained network and an input data layer, and then"
            " extract features of the input data produced by the net.\n"
            "Usage: extract_features pretrained_net_param"
            " feature_extraction_proto_file extract_feature_blob_name1[,name2,...]"
            " save_feature_dataset_name1[,name2,...] num_mini_batches db_type"
    }
}

```

```

" [CPU/GPU] [DEVICE_ID=0]\n"
"Note: you can extract multiple features in one pass by specifying"
" multiple feature blob names and dataset names seperated by ','."
" The names cannot contain white space characters and the number of blobs"
" and datasets must be equal.";
return 1;
}

int arg_pos = num_required_args;

arg_pos = num_required_args; //重复代码...
// 获得 CPU/GPU、GPU ID 等命令行参数值
if (argc > arg_pos && strcmp(argv[arg_pos], "GPU") == 0) {
    LOG(ERROR) << "Using GPU";
    uint device_id = 0;
    if (argc > arg_pos + 1) {
        device_id = atoi(argv[arg_pos + 1]);
        CHECK_GE(device_id, 0);
    }
    LOG(ERROR) << "Using Device_id=" << device_id;
    Caffe::SetDevice(device_id);
    Caffe::set_mode(Caffe::GPU);
} else {
    LOG(ERROR) << "Using CPU";
    Caffe::set_mode(Caffe::CPU);
}

arg_pos = 0;
std::string pretrained_binary_proto(argv[++arg_pos]); // 获得预训练模型的文件名 (*.caffemodel)

// 获得网络描述文件名 (*.prototxt)
std::string feature_extraction_proto(argv[++arg_pos]);
// 用 *.prototxt 创建一个 Net 对象, 设置当前为测试阶段
shared_ptr<Net<Dtype>> feature_extraction_net(
    new Net<Dtype>(feature_extraction_proto, caffe::TEST));
// 从预训练模型 *.caffemodel 拷贝 weight、bias 到新建的 Net 中
feature_extraction_net->CopyTrainedLayersFrom(pretrained_binary_proto);
// 获取待提取特征所在 Blob 的名称
std::string extract_feature_blob_names(argv[++arg_pos]);
std::vector<std::string> blob_names;

```

```
boost::split(blob_names, extract_feature_blob_names, boost::is_any_of(",")); // 将命令
// 行输入的名称转换为字符串 vector 数组
```

```
// 获取用于保存特征的文件名
```

```
std::string save_feature_dataset_names(argv[++arg_pos]);
```

```
std::vector<std::string> dataset_names;
```

```
boost::split(dataset_names, save_feature_dataset_names,
```

```
    boost::is_any_of(",")); // 将命令行输入名称转换为字符串 vector 数组
```

```
// 确保两个数组维度相同
```

```
CHECK_EQ(blob_names.size(), dataset_names.size()) <<
```

```
    " the number of blob names and dataset names must be equal";
```

```
size_t num_features = blob_names.size();
```

```
// 确保新建的网络中包含待提取特征的 Blob 名称, 否则报错
```

```
for (size_t i = 0; i < num_features; i++) {
```

```
    CHECK(feature_extraction_net->has_blob(blob_names[i]))
```

```
    << "Unknown feature blob name " << blob_names[i]
```

```
    << " in the network " << feature_extraction_proto;
```

```
}
```

```
// 获得批量数目, 而每个批量包含的图片数在*.prototxt 中的 data_layer 参数中指定
```

```
int num_mini_batches = atoi(argv[++arg_pos]);
```

```
std::vector<shared_ptr<db::DB>> feature_dbs;
```

```
std::vector<shared_ptr<db::Transaction>> txns;
```

```
const char* db_type = argv[++arg_pos];
```

```
for (size_t i = 0; i < num_features; ++i) {
```

```
    LOG(INFO) << "Opening dataset " << dataset_names[i];
```

```
    shared_ptr<db::DB> db(db::GetDB(db_type));
```

```
    db->Open(dataset_names.at(i), db::NEW);
```

```
    feature_dbs.push_back(db);
```

```
    shared_ptr<db::Transaction> txn(db->NewTransaction());
```

```
    txns.push_back(txn);
```

```
}
```

```
LOG(ERROR) << "Extracting Features";
```

```
Datum datum;
```

```
const int kMaxKeyStrLength = 100;
```

```
char key_str[kMaxKeyStrLength];
```

```
std::vector<Blob<float>*> input_vec;
```

```
std::vector<int> image_indices(num_features, 0);
```

```
for (int batch_index = 0; batch_index < num_mini_batches; ++batch_index) {
```

```

feature_extraction_net->Forward(input_vec);
for (int i = 0; i < num_features; ++i) {
    const shared_ptr<Blob<Dtype>> feature_blob = feature_extraction_net
        ->blob_by_name(blob_names[i]);
    int batch_size = feature_blob->num();
    int dim_features = feature_blob->count() / batch_size;
    const Dtype* feature_blob_data;
    for (int n = 0; n < batch_size; ++n) {
        datum.set_height(feature_blob->height());
        datum.set_width(feature_blob->width());
        datum.set_channels(feature_blob->channels());
        datum.clear_data();
        datum.clear_float_data();
        feature_blob_data = feature_blob->cpu_data() +
            feature_blob->offset(n);
        for (int d = 0; d < dim_features; ++d) {
            datum.add_float_data(feature_blob_data[d]);
        }
        int length = snprintf(key_str, kMaxKeyStrLength, "%010d",
            image_indices[i]);
        string out;
        CHECK(datum.SerializeToString(&out));
        txns.at(i)->Put(std::string(key_str, length), out);
        ++image_indices[i];
        if (image_indices[i] % 1000 == 0) {
            txns.at(i)->Commit();
            txns.at(i).reset(feature_dbs.at(i)->NewTransaction());
            LOG(ERROR)<< "Extracted features of " << image_indices[i] <<
                " query images for feature blob " << blob_names[i];
        }
    } // for (int n = 0; n < batch_size; ++n)
} // for (int i = 0; i < num_features; ++i)
} // for (int batch_index = 0; batch_index < num_mini_batches; ++batch_index)
// 写最后一批
for (int i = 0; i < num_features; ++i) {
    if (image_indices[i] % 1000 != 0) {
        txns.at(i)->Commit();
    }
    LOG(ERROR)<< "Extracted features of " << image_indices[i] <<

```

```

    " query images for feature blob " << blob_names[i];
    feature_dbs.at(i)->Close();
}

LOG(ERROR)<< "Successfully extracted the features!";
return 0;
}

```

14.3 转换图像格式

在 Caffe 数据输入层所有数据都以 LEVELDB 或 LMDB 形式提供。为此需要在预处理阶段将不同数据集转换为 LEVELDB 或 LMDB 格式。第 6 天内容已经介绍了 MNIST 数据集图像转换的过程。今天我们继续研读 CIFAR10 和 ImageNet 两个数据集的图像格式转换过程。

CIFAR10 图像格式转换程序位于 `examples/cifar10/convert_cifar_data.cpp`:

```

//
// 该程序将 CIFAR 数据集转换为 Caffe 需要的格式
// 用法:
//   convert_cifar_data input_folder output_db_file
// CIFAR 数据集从这里下载:
//   http://www.cs.toronto.edu/~kriz/cifar.html

#include <fstream> // NOLINT(readability/streams)
#include <string>

#include "boost/scoped_ptr.hpp"
#include "glog/logging.h"
#include "google/protobuf/text_format.h"
#include "stdint.h"

#include "caffe/proto/caffe.pb.h"
#include "caffe/util/db.hpp"

using caffe::Datum;
using boost::scoped_ptr;
using std::string;
namespace db = caffe::db;
// CIFAR 数据集图像尺寸为 32 x 32

```

```

const int kCIFARSize = 32;
const int kCIFARImageNBytes = 3072;
const int kCIFARBatchSize = 10000;
const int kCIFARTrainBatches = 5;
// 从文件 file 读取图像数据 到 buffer 和 label 缓冲区
void read_image(std::ifstream* file, int* label, char* buffer) {
    char label_char;
    file->read(&label_char, 1);
    *label = label_char;
    file->read(buffer, kCIFARImageNBytes);
    return;
}
// 转换图像格式核心代码
void convert_dataset(const string& input_folder, const string& output_folder,
    const string& db_type) {
    scoped_ptr<db::DB> train_db(db::GetDB(db_type)); // 创建 db 句柄
    // 打开 db
    train_db->Open(output_folder + "/cifar10_train_" + db_type, db::NEW);
    scoped_ptr<db::Transaction> txn(train_db->NewTransaction());
    // 数据缓冲区, 用于读取一张 CIFAR 图片和对应标签
    int label;
    char str_buffer[kCIFARImageNBytes];
    Datum datum;
    datum.set_channels(3);
    datum.set_height(kCIFARSize);
    datum.set_width(kCIFARSize);
    // 写训练数据集
    LOG(INFO) << "Writing Training data";
    for (int fileid = 0; fileid < kCIFARTrainBatches; ++fileid) {
        // 读取文件
        LOG(INFO) << "Training Batch " << fileid + 1;
        snprintf(str_buffer, kCIFARImageNBytes, "/data_batch_%d.bin", fileid + 1);
        std::ifstream data_file((input_folder + str_buffer).c_str(),
            std::ios::in | std::ios::binary);
        CHECK(data_file) << "Unable to open train file #" << fileid + 1;
        for (int itemid = 0; itemid < kCIFARBatchSize; ++itemid) {
            read_image(&data_file, &label, str_buffer); // 从文件读入数据缓冲区
            datum.set_label(label); // 记录标签
            datum.set_data(str_buffer, kCIFARImageNBytes); // 记录图像数据
        }
    }
}

```



```

    int length = snprintf(str_buffer, kCIFARImageNBytes, "%05d",
        fileid * kCIFARBatchSize + itemid);
    string out;
    CHECK(datum.SerializeToString(&out)); // 序列化为字符串
    txn->Put(string(str_buffer, length), out); // 写入 db
}

}

txn->Commit();
train_db->Close();
// 写测试数据集, 过程与写训练数据集类似
LOG(INFO) << "Writing Testing data";
scoped_ptr<db::DB> test_db(db::GetDB(db_type));
test_db->Open(output_folder + "/cifar10_test_" + db_type, db::NEW);
txn.reset(test_db->NewTransaction());
// 打开文件
std::ifstream data_file((input_folder + "/test_batch.bin").c_str(),
    std::ios::in | std::ios::binary);
CHECK(data_file) << "Unable to open test file.";
for (int itemid = 0; itemid < kCIFARBatchSize; ++itemid) {
    read_image(&data_file, &label, str_buffer);
    datum.set_label(label);
    datum.set_data(str_buffer, kCIFARImageNBytes);
    int length = snprintf(str_buffer, kCIFARImageNBytes, "%05d", itemid);
    string out;
    CHECK(datum.SerializeToString(&out));
    txn->Put(string(str_buffer, length), out);
}

txn->Commit();
test_db->Close();
}

// 主函数
int main(int argc, char** argv) {
    if (argc != 4) {
        printf("This script converts the CIFAR dataset to the leveldb format used\n"
            "by caffe to perform classification.\n"
            "Usage:\n"
            "    convert_cifar_data input_folder output_folder db_type\n"
            "Where the input folder should contain the binary batch files.\n"
            "The CIFAR dataset could be downloaded at\n");
    }
}

```

```

    "    http://www.cs.toronto.edu/~kriz/cifar.html\n"
    "You should gunzip them after downloading.\n");
} else {
    google::InitGoogleLogging(argv[0]);
    convert_dataset(string(argv[1]), string(argv[2]), string(argv[3]));
}
return 0;
}

```

ImageNet 数据集转换程序位于 tools/convert_imageset.cpp 中：

```

// 该程序将一组图像集转换为 lmdb/leveldb 格式
// 用法:
//  convert_imageset [FLAGS] ROOTFOLDER/ LISTFILE DB_NAME
//
// 上面命令行参数 ROOTFOLDER 为存放所有图片的根目录
// LISTFILE 是一个文本文件，记录了图片文件与对应标签的映射关系，格式为：
//  subfolder1/file1.JPEG 7
// 每行一条记录
//
#include <algorithm>
#include <fstream> // NOLINT(readability/streams)
#include <string>
#include <utility>
#include <vector>

#include "boost/scoped_ptr.hpp"
#include "gflags/gflags.h"
#include "glog/logging.h"

#include "caffe/proto/caffe.pb.h"
#include "caffe/util/db.hpp"
#include "caffe/util/io.hpp"
#include "caffe/util/rng.hpp"

using namespace caffe; // NOLINT(build/namespaces)
using std::pair;
using boost::scoped_ptr;

```

```

DEFINE_bool(gray, false,
    "When this option is on, treat images as grayscale ones");
DEFINE_bool(shuffle, false,
    "Randomly shuffle the order of images and their labels");
DEFINE_string(backend, "lmdb",
    "The backend {lmdb, leveldb} for storing the result");
DEFINE_int32(resize_width, 0, "Width images are resized to");
DEFINE_int32(resize_height, 0, "Height images are resized to");
DEFINE_bool(check_size, false,
    "When this option is on, check that all the datum have the same size");
DEFINE_bool(encoded, false,
    "When this option is on, the encoded image will be save in datum");
DEFINE_string(encode_type, "",
    "Optional: What type should we encode the image as ('png', 'jpg', ...).");

int main(int argc, char** argv) {
    ::google::InitGoogleLogging(argv[0]);

#ifdef GFLAGS_GFLAGS_H_
    namespace gflags = google;
#endif

    gflags::SetUsageMessage("Convert a set of images to the leveldb/lmdb\n"
        "format used as input for Caffe.\n"
        "Usage:\n"
        "    convert_imageset [FLAGS] ROOTFOLDER/ LISTFILE DB_NAME\n"
        "The ImageNet dataset for the training demo is at\n"
        "    http://www.image-net.org/download-images\n");
    gflags::ParseCommandLineFlags(&argc, &argv, true);

    if (argc < 4) {
        gflags::ShowUsageWithFlagsRestrict(argv[0], "tools/convert_imageset");
        return 1;
    }

    const bool is_color = !FLAGS_gray;
    const bool check_size = FLAGS_check_size;
    const bool encoded = FLAGS_encoded;
    const string encode_type = FLAGS_encode_type;

```

```

std::ifstream infile(argv[2]);
std::vector<std::pair<std::string, int>> lines;
std::string filename;
int label;
while (infile >> filename >> label) {
    lines.push_back(std::make_pair(filename, label));
}
if (FLAGS_shuffle) {
    // 随机打乱数据
    LOG(INFO) << "Shuffling data";
    shuffle(lines.begin(), lines.end());
}
LOG(INFO) << "A total of " << lines.size() << " images.";

if (encode_type.size() && !encoded)
    LOG(INFO) << "encode_type specified, assuming encoded=true.";

int resize_height = std::max<int>(0, FLAGS_resize_height);
int resize_width = std::max<int>(0, FLAGS_resize_width);

// 创建新的 db
scoped_ptr<db::DB>db(db::GetDB(FLAGS_backend));
db->Open(argv[3], db::NEW);
scoped_ptr<db::Transaction>txn(db->NewTransaction());

// 保存至 db
std::string root_folder(argv[1]);
Datum datum;
int count = 0;
const int kMaxKeyLength = 256;
char key_cstr[kMaxKeyLength];
int data_size = 0;
bool data_size_initialized = false;

for (int line_id = 0; line_id < lines.size(); ++line_id) {
    bool status;
    std::string enc = encode_type;
    if (encoded && !enc.size()) {

```

```

// 从文件名猜测编码方式
string fn = lines[line_id].first;
size_t p = fn.rfind('.');
if ( p == fn.npos )
    LOG(WARNING) << "Failed to guess the encoding of '" << fn << "'";
enc = fn.substr(p);
std::transform(enc.begin(), enc.end(), enc.begin(), ::tolower);
}
status = ReadImageToDatum(root_folder + lines[line_id].first,
    lines[line_id].second, resize_height, resize_width, is_color,
    enc, &datum);
if (status == false) continue;
if (check_size) {
    if (!data_size_initialized) {
        data_size = datum.channels() * datum.height() * datum.width();
        data_size_initialized = true;
    } else {
        const std::string& data = datum.data();
        CHECK_EQ(data.size(), data_size) << "Incorrect data field size "
            <<data.size();
    }
}
// 序列化
int length = snprintf(key_cstr, kMaxKeyLength, "%08d_%s", line_id,
    lines[line_id].first.c_str());

// 写入 db
string out;
CHECK(datum.SerializeToString(&out));
txn->Put(string(key_cstr, length), out);

if (++count % 1000 == 0) {
    // 提交
    txn->Commit();
    txn.reset(db->NewTransaction());
    LOG(ERROR) << "Processed " << count << " files.";
}
}
// 写最后一批数据

```

```

if (count % 1000 != 0) {
    txn->Commit();
    LOG(ERROR) << "Processed " << count << " files.";
}
return 0;
}

```

14.4 计算图像均值

在数据读取层的 Transform 阶段，需要去均值操作。均值文件一般需要用原始数据计算得到，本节将介绍 Caffe 中这一工具的实现，其位于 tools/compute_image_mean.cpp。

```

#include <stdint.h>
#include <algorithm>
#include <string>
#include <utility>
#include <vector>

#include "boost/scoped_ptr.hpp"
#include "gflags/gflags.h"
#include "glog/logging.h"

#include "caffe/proto/caffe.pb.h"
#include "caffe/util/db.hpp"
#include "caffe/util/io.hpp"

using namespace caffe; // NOLINT(build/namespaces)

using std::max;
using std::pair;
using boost::scoped_ptr;
// 命令行可以指定使用 lmdb 或 leveldb 作为输入图像源
DEFINE_string(backend, "lmdb",
    "The backend {leveldb, lmdb} containing the images");

int main(int argc, char** argv) {
    ::google::InitGoogleLogging(argv[0]); // 初始化 GLOG

#ifdef GFLAGS_GFLAGS_H_

```

```

namespace gflags = google;
#endif

// 设置 GFLAGS 命令行提示信息
gflags::SetUsageMessage("Compute the mean_image of a set of images given by"
    " a leveldb/lmdb\n"
    "Usage:\n"
    "    compute_image_mean [FLAGS] INPUT_DB [OUTPUT_FILE]\n");

gflags::ParseCommandLineFlags(&argc, &argv, true); // 解析命令行参数

if (argc < 2 || argc > 3) {
    gflags::ShowUsageWithFlagsRestrict(argv[0], "tools/compute_image_mean");
    return 1;
}

scoped_ptr<db::DB> db(db::GetDB(FLAGS_backend)); // 获得输入数据的 db 类型，并创建对象
db->Open(argv[1], db::READ); // 以只读方式打开 db 文件
scoped_ptr<db::Cursor> cursor(db->NewCursor()); // 创建 db 指针

BlobProto sum_blob; // 求和、取平均就靠它了
int count = 0;
// 获取第一个 db 数据
Datum datum;
datum.ParseFromString(cursor->value());

if (DecodeDatumNative(&datum)) {
    LOG(INFO) << "Decoding Datum";
}

// sum_blob 尺寸为 1 x C x H x W
sum_blob.set_num(1);
sum_blob.set_channels(datum.channels());
sum_blob.set_height(datum.height());
sum_blob.set_width(datum.width());
const int data_size = datum.channels() * datum.height() * datum.width();
int size_in_datum = std::max<int>(datum.data().size(),
    datum.float_data_size());

// 初始化数据为 0
for (int i = 0; i < size_in_datum; ++i) {
    sum_blob.add_data(0.);
}

```

```

}
LOG(INFO) << "Starting Iteration";
while (cursor->valid()) { // 开始大循环
    Datum datum;
    datum.ParseFromString(cursor->value()); // 获得一个 datum
    DecodeDatumNative(&datum); // 解码

    const std::string& data = datum.data();
    size_in_datum = std::max<int>(datum.data().size(),
        datum.float_data_size());
    CHECK_EQ(size_in_datum, data_size) << "Incorrect data field size " <<
        size_in_datum;
    if (data.size() != 0) {
        CHECK_EQ(data.size(), size_in_datum);
        for (int i = 0; i < size_in_datum; ++i) {
            sum_blob.set_data(i, sum_blob.data(i) + (uint8_t)data[i]);
        }
    } else {
        CHECK_EQ(datum.float_data_size(), size_in_datum);
        for (int i = 0; i < size_in_datum; ++i) {
            sum_blob.set_data(i, sum_blob.data(i) +
                static_cast<float>(datum.float_data(i)));
        }
    }
    ++count;
    if (count % 10000 == 0) {
        LOG(INFO) << "Processed " << count << " files.";
    }
    cursor->Next();
}

if (count % 10000 != 0) {
    LOG(INFO) << "Processed " << count << " files.";
}
for (int i = 0; i < sum_blob.data_size(); ++i) {
    sum_blob.set_data(i, sum_blob.data(i) / count);
}
// 写到磁盘，以二进制 ProtoBuffer 文件格式保存
if (argc == 3) {

```



```

LOG(INFO) << "Write to " << argv[2];
WriteProtoToBinaryFile(sum_blob, argv[2]);
}
const int channels = sum_blob.channels();
const int dim = sum_blob.height() * sum_blob.width();
std::vector<float> mean_values(channels, 0.0);
LOG(INFO) << "Number of channels: " << channels;
for (int c = 0; c < channels; ++c) {
    for (int i = 0; i < dim; ++i) {
        mean_values[c] += sum_blob.data(dim * c + i);
    }
    LOG(INFO) << "mean_value channel [" << c << "]: " << mean_values[c] / dim;
}
return 0;
}

```

14.5 自己编写工具

无论什么工具，最初的出发点都是为了满足某个需求。这个需求可能来自客户、老板、同事、导师、网友求助，然而最大的需求方往往是开发者自己。经常思考并针对自己实际工作学习中遇到的各类困惑，创造性地编写一些工具能大大提高效率并触发更多灵感，促进对深度学习理论和 Caffe 代码框架的理解。通过今天的学习，读者可以尝试记录下每天自己的困惑，转化为具体需求并用代码实现，你将有希望成长为优秀的产品经理。

14.6 练习题

1. 写一个简单的权值抽取工具，从已经训练好的 Caffe 模型导出权值到文件。
2. 写一个计算已经训练好的 Caffe 模型权值最大值、最小值、平均值的工具。

3. 前面几个工具都只是单线程的应用程序，每次启动都需要经历构建网络、初始化权值等过程，不适合实际互联网在线系统。为了重用网络，需要将其常驻内存，成为服务，当有请求时（例如用户上传一张图片）触发一次过网络操作。读者可以尝试在特征提取例程基础上进行修改，得到服务化的 Caffe。如果对 Web 服务不熟悉，可以参考 NVIDIA DIGITS 工具的源码（<https://developer.nvidia.com/digits>）。

篇尾语

中篇·热恋主要内容是深入源码近距离观察了 Caffe 内部运作方式，了解了深度学习的一些基础概念如何映射为代码实现，拉近了程序员与科学家的距离。从事科研工作的读者可以从中学习如何将自己的想法转化为代码实现，而从事一线开发的程序员读者可以从中学习代码背后的理论知识，充实自己的算法设计。如果读者看完本篇仍意犹未尽，不如 pull 最新源码，结合本篇的方法，再从头阅读一遍枝干代码，相信会有驾轻就熟之感。

本篇介绍了几种阅读代码的策略，可以有效帮助读者应付绝大多数框架。简单来说，有数据流跟踪、函数栈跟踪、日志跟踪、调试跟踪等几种方法，可以任意组合发挥其各自功效，相信读者在阅读其他框架时使用这些方式能游刃有余。

无论我们从事哪方面工作，学习的目的都是为了获得自由。掌握了 C++即获得阅读 Caffe 代码的自由；掌握了 Caffe 实现即获得使用深度学习框架的自由；掌握了深度学习理论即获得数据选取、模型调参的自由；掌握了 GPU/CUDA 即获得计算加速的自由……随着学习的深入，相信你会不断扩大自由活动空间，最终达到财务自由。

下篇 升华

曾经沧海难为水，除却巫山不是云。

取次花丛懒回顾，半缘修道半缘君。

寻常百种花齐发，偏摘梨花与白人。

今日江头两三树，可怜和叶度残春。

——元稹《离思五首·其四》

上篇和中篇对于读者具体使用硬件平台没有限制，一台普通笔记本电脑就能完成前面所有实验。但从本篇开始，我们就要考虑实际生产和应用环境下如何部署 Caffe，追求高性能、快速迁移部署、特定应用下的适配。

第 15 天

Caffe 计算加速

通过上篇和中篇我们了解了 Caffe 的使用和具体实现细节，但一直是在 CPU 上运行，速度较慢，只适合跑一些小模型。今天我们体验一下 Caffe GPU/cuDNN 加速模式。掌握利用 GPU 加速深度学习或其他问题的方法，可以让你成长为优秀的软件优化工程师。这个岗位一般由算法工程师兼任，实际上算法工程师更多关注某个领域算法在理论方面的优化，而软件优化工程师则与领域相关度不高，只是根据实际硬件架构合理调整算法步骤和语句，同时保证算法输入输出不变，使软件在特定硬件上达到最高运算效率。在大多数企业中该类人才缺口非常大。

15.1 Caffe 计时功能

编译好的 Caffe 可以通过运行 `caffe time` 命令，对当前平台（目前是 CPU）上网络各层前向/后向计算进行计时：

```
$ ./build/tools/caffe.bin time \
-model examples/mnist/lenet_train_test.prototxt

I0405 14:52:34.514957 1965830144 caffe.cpp:312] Use CPU.
I0405 14:52:34.520442 1965830144 net.cpp:313] The NetState phase (0) differed from the
phase (1) specified by a rule in layer mnist
I0405 14:52:34.520478 1965830144 net.cpp:313] The NetState phase (0) differed from the
phase (1) specified by a rule in layer accuracy
I0405 14:52:34.520488 1965830144 net.cpp:49] Initializing net from parameters:
// .....网络描述略
I0405 14:52:34.520659 1965830144 layer_factory.hpp:77] Creating layer mnist
// .....各层描述略
I0405 14:52:34.535886 1965830144 caffe.cpp:320] Performing Forward
```

```

I0405 14:52:34.562958 1965830144 caffe.cpp:325] Initial loss: 2.4196
I0405 14:52:34.562996 1965830144 caffe.cpp:326] Performing Backward
I0405 14:52:34.595528 1965830144 caffe.cpp:334] *** Benchmark begins ***
I0405 14:52:34.595584 1965830144 caffe.cpp:335] Testing for 50 iterations.
I0405 14:52:34.645946 1965830144 caffe.cpp:363] Iteration: 1 forward-backward time: 50 ms.
// 迭代次数 2~49 略
I0405 14:52:37.178436 1965830144 caffe.cpp:363] Iteration: 50 forward-backward time: 50 ms.
// 对每个层进行计时
I0405 14:52:37.178477 1965830144 caffe.cpp:366] Average time per layer:
I0405 14:52:37.178486 1965830144 caffe.cpp:369]      mnist      forward: 0.038 ms.
I0405 14:52:37.178504 1965830144 caffe.cpp:372]      mnist      backward: 0.00102 ms.
I0405 14:52:37.178514 1965830144 caffe.cpp:369]      conv1      forward: 5.45956 ms.
I0405 14:52:37.178520 1965830144 caffe.cpp:372]      conv1      backward: 2.69554 ms.
I0405 14:52:37.178527 1965830144 caffe.cpp:369]      pool1      forward: 5.42452 ms.
I0405 14:52:37.178534 1965830144 caffe.cpp:372]      pool1      backward: 2.11508 ms.
I0405 14:52:37.178541 1965830144 caffe.cpp:369]      conv2      forward: 8.9766 ms.
I0405 14:52:37.178549 1965830144 caffe.cpp:372]      conv2      backward: 19.0001 ms.
I0405 14:52:37.178555 1965830144 caffe.cpp:369]      pool2      forward: 3.19856 ms.
I0405 14:52:37.178562 1965830144 caffe.cpp:372]      pool2      backward: 1.66524 ms.
I0405 14:52:37.178568 1965830144 caffe.cpp:369]      ip1        forward: 0.93266 ms.
I0405 14:52:37.178575 1965830144 caffe.cpp:372]      ip1        backward: 1.64174 ms.
I0405 14:52:37.178581 1965830144 caffe.cpp:369]      relu1      forward: 0.04648 ms.
I0405 14:52:37.178588 1965830144 caffe.cpp:372]      relu1      backward: 0.06176 ms.
I0405 14:52:37.178594 1965830144 caffe.cpp:369]      ip2        forward: 0.07234 ms.
I0405 14:52:37.178601 1965830144 caffe.cpp:372]      ip2        backward: 0.18076 ms.
I0405 14:52:37.178608 1965830144 caffe.cpp:369]      loss       forward: 0.04114 ms.
I0405 14:52:37.178614 1965830144 caffe.cpp:372]      loss       backward: 0.00178 ms.
// 平均前向传播计算时间
I0405 14:52:37.178624 1965830144 caffe.cpp:377] Average Forward pass: 24.2216 ms.
// 平均反向传播计算时间
I0405 14:52:37.178630 1965830144 caffe.cpp:379] Average Backward pass: 27.3904 ms.
// 平均前向+反向传播计算时间
I0405 14:52:37.178637 1965830144 caffe.cpp:381] Average Forward-Backward: 51.66 ms.
// 50 次迭代总时间
I0405 14:52:37.178643 1965830144 caffe.cpp:383] Total Time: 2583 ms.
// 结束
I0405 14:52:37.178650 1965830144 caffe.cpp:384] *** Benchmark ends ***

```

利用 Caffe 的计时功能，可以对比不同硬件、不同算法、不同模型的处理耗时情况，指导运维工程师、算法工程师和模型设计师有针对性地进行硬件/算法/模型选型和评估。

15.2 Caffe GPU 加速模式

直白点说,本节将介绍的 GPU 就是电脑上的显卡,打游戏时画面卡顿往往与显卡性能有关。提高电脑显卡配置可以提升游戏体验,但代价是需要大功率的电源和散热系统。读者可选择租用 GPU 云服务器(如阿里云 HPC, <https://www.aliyun.com/product/hpc>)来满足本节使用 GPU 加速 Caffe 的需求。

15.2.1 GPU 是什么

从 1965 年开始,计算机和处理器一直按照摩尔定律不断提高性能,其中主频的提升发挥了主要作用。而 2004 年左右,晶体管的功耗问题成为主要瓶颈,难以支撑更高的主频。与此同时,多核处理器应运而生,延续了摩尔定律。为了充分利用 CPU 的计算资源,越来越多的算法被重新设计成并行结构,以适应多核 CPU 的架构。

GPU (Graphics Processing Unit) 即图形处理器,主要承担 2D 或 3D 图形处理任务,最初用作纹理映射和多边形着色等基本计算机图形任务。近年的 GPU 拥有可编程着色器,能够协助 CPU 完成过采样、插值、色彩空间变换等计算任务。GPU 的并行架构,天然具备向量处理、矩阵计算功能。

图 15-1 显示了 2002 年以来 Intel CPU 与 NVIDIA GPU 计算能力的发展情况。

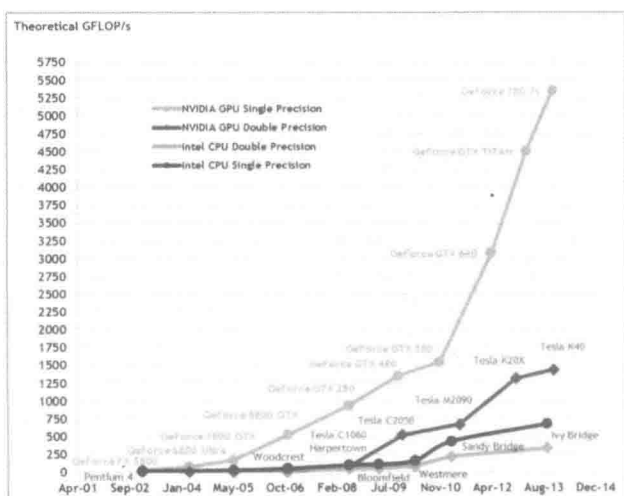


图 15-1 CPU 和 GPU 浮点处理能力对比^[1]

从图 15-1 看出, GPU 的计算能力发展速度远远超过了同时期的 CPU, 一些并行计算任务在 GPU 上可以获得显著加速。图 15-2 显示了部分 NVIDIA GPU 型号。

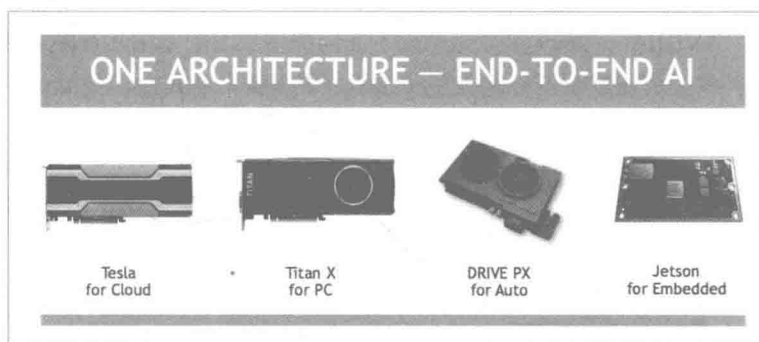


图15-2 NVIDIA的不同型号GPU

其中, Tesla 系列 GPU 适用于云端部署, 具有较高的稳定性和可靠性; Titan X 为消费级显卡, 性价比高, 适合 PC 上使用; Jetson 系列 GPU 适合嵌入式应用。

更多 GPU 信息, 请参考 NVIDIA 官网。

15.2.2 CUDA 是什么

CUDA (Compute Unified Device Architecture) 是由 NVIDIA 在 2006 年推出的一套针对异构计算资源(说白了就是 Intel CPU + 自家 GPU)下的大规模并行计算的架构, 包括编译器(nvcc)、开发工具、运行时库和驱动等模块, 是当今最流行的 GPU 编程环境。

在 CUDA 之前, 一些极客使用计算机图形学语言 (Open GL、Shader 语言) 实现算法并在 GPU 上运行, 普通用户难以掌握其高超的技巧。

CUDA 语法和 C 语言高度相似, 大大降低了 GPU 编程门槛。

与 CUDA 同时期的还有 Open CL 语言标准, 支持 CPU/GPU/FPGA/DSP/ASIC 等异构平台, 但相应软件工具发展缓慢, 存在效率不高、软件库缺失等问题, 期待不久的将来有大的进步。

15.2.3 GPU、CUDA 和深度学习

我们从第 2 天就了解到, 深度学习无论是 CNN/DNN/RNN, 算法均涉及大量矩阵-向量乘运算, 具有内在并行性, 适合在 GPU 上实现。

幸运的是，我们不需要手动编写 CUDA 代码，而是直接利用 CUDA 中提供的 cuBLAS 软件库，其作用相当于 CPU 上的 OpenBLAS/MKL 计算库。

更幸运的是，NVIDIA 从 2014 年开始推出了面向深度学习的专用加速库 cuDNN^[2]，至今已有 5 个版本，支持常见的深度学习计算类型（卷积、下采样、非线性、Softmax）。图 15-3 显示了 NVIDIA 为深度学习用户提供的全套软硬件解决方案。

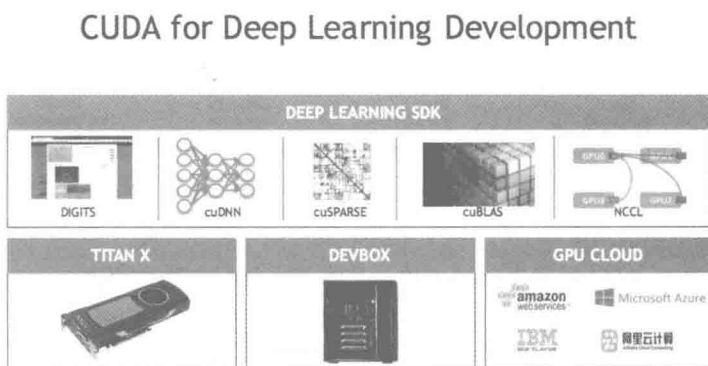


图15-3 NVIDIA适用于深度学习的软硬件解决方案

从图 15-3 中可见，软件包括深度学习 SDK（DIGITS、cuDNN、cuSPARSE、cuBLAS、NCCL），而硬件则可选择消费级器件（Titan X）、专业硬件系统（DEVBOX）或者云上的解决方案（AWS、Azure、IBM SoftLayer、阿里云 HPC，<https://www.aliyun.com/product/hpc>）。

深度学习社区也开发了特定 GPU 架构上的 CNN/DNN 实现，如 cuda-convnet2^[3]、maxDNN^[4]。Caffe 源码中所有 *.cu 文件均使用 CUDA 及其软件库编写，而且与 CPU 代码一一对照，通过这些代码可以让读者快速学习基于 CUDA 的 GPU 编程方法。

15.2.4 Caffe GPU 环境准备

为了让 Caffe 支持 GPU 模式，需要安装 GPU 驱动和 CUDA Toolkit，请新手尽量不要看网上的各种教程，而是直接阅读 NVIDIA 官方安装手册 *NVIDIA CUDA Installation Guide For Linux/Mac OS X/Windows*^[5]，这样可以少踩坑。

使用阿里云 HPC 物理机，交付时已安装 GPU 驱动 352.79 和 CUDA Toolkit 7.5，安装路径为默认路径 /usr/local/cuda/，其中用于编译 GPU CUDA 代码的编译器 nvcc 位于 /usr/local/cuda/bin/nvcc。如果读者是 CUDA 初学者，建议首先从 /usr/local/cuda/samples 开始学习

如何编译和运行 GPU 代码。本节限于篇幅,不再赘述。CUDA 入门资料亦可参考笔者 2012 年的博客^[6]。

1. 验证驱动安装成功

在命令行执行 NVIDIA 系统管理接口命令 `nvidia-smi`, 可以看到 GPU 设备和驱动以及占用率等详情, 如图 15-4 所示。

```
# nvidia-smi
```

```
Fri Apr 8 09:28:29 2016
```

NVIDIA-SMI 352.79 Driver Version: 352.79									
GPU	Name	Persistence-MI	Bus-Id	Disp.A	Volatile Uncorr. ECC				
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.			
0	Tesla K40m	Off	0000:02:00.0	Off					
N/A	22C	P8	19W / 235W	61MiB / 11519MiB	0%	Default			
1	Tesla K40m	Off	0000:03:00.0	Off					
N/A	24C	P8	20W / 235W	61MiB / 11519MiB	0%	Default			

Processes:					GPU Memory
GPU	PID	Type	Process name	Usage	
0	7542	G	Xorg		4MiB
1	7542	G	Xorg		4MiB

图15-4 Linux下nvidia-smi详情

该命令十分强大, 可以控制 GPU 超频, 监控 GPU 运行信息, 感兴趣的读者可以运行“`nvidia-smi -h`”获得详细命令清单。

在 Windows 下 NVSMI 工具默认安装位置为 `C:\Program Files\NVIDIA Corporation\NVSMI\` `nvidia-smi.exe`, 运行效果如图 15-5 所示。

```
C:\Windows\system32\cmd.exe
```

GPU	PID	Type	Process name	Usage
0	948	C	...caffe-master\Build\x64\Release\caffe.exe	210MiB

```
Sun Mar 27 19:37:10 2016
```

NVIDIA-SMI 354.70 Driver Version: 354.70									
GPU	Name	TCC/UDSM	Bus-Id	Disp.B	Volatile Uncorr. ECC				
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.			
0	Tesla M40	TCC=	0000:03:00.0	Off					
0%	33C	P8	118W / 250W	269MiB / 11519MiB	89%	Default			
1	Tesla M40	TCC=	0000:03:00.0	Off					
0%	25C	P8	17W / 250W	58MiB / 11519MiB	0%	Default			

Processes:					GPU Memory
GPU	PID	Type	Process name	Usage	
0	948	C	...caffe-master\Build\x64\Release\caffe.exe	210MiB	

图15-5 Windows下nvidia-smi详情

2. 验证 CUDA 安装成功

我们这里运行一个 cuBLAS 例程来验证 CUDA 环境安装正常。注意需要 root 权限。首先进入例程所在目录：

```
# cd /usr/local/cuda/samples/7_CUDALibraries/batchCUBLAS/
```

查看当前目录下文件：

```
# ls
batchCUBLAS.cpp batchCUBLAS.h Makefile NsightEclipse.xml readme.txt
```

该例程提供了 Makefile，可以直接运行 make 编译：

```
# make
/usr/local/cuda-7.5/bin/nvcc -ccbin g++ -I../common/inc -m64 -gencode
arch=compute_52,code=sm_52 -gencode arch=compute_52,code=compute_52 -o batchCUBLAS.o -c
batchCUBLAS.cpp
/usr/local/cuda-7.5/bin/nvcc -ccbin g++ -m64 -gencode arch=compute_52,code=sm_52
-gencode arch=compute_52,code=compute_52 -o batchCUBLAS batchCUBLAS.o -lcublas
mkdir -p ../../bin/x86_64/linux/release
cp batchCUBLAS ../../bin/x86_64/linux/release
```

可以看到编译该例程时命令行调用了 nvcc 编译器（默认路径为 /usr/local/cuda-7.5/bin/nvcc），读者可以通过阅读文档^[7]学习其命令行参数的具体意义。再次查看当前目录下文件，发现可执行文件 batchCUBLAS：

```
# ls
batchCUBLAS batchCUBLAS.cpp batchCUBLAS.h batchCUBLAS.o Makefile NsightEclipse.xml
readme.txt
```

这是一个利用 cuBLAS 实现矩阵乘计算的例程，我们这样运行：

```
# ./batchCUBLAS -m4096 -n4096 -k4096
batchCUBLAS Starting...
GPU Device 0: "Tesla M40" with compute capability 5.2
==== Running single kernels ====
Testing sgemm
#### args: ta=0 tb=0 m=4096 n=4096 k=4096 alpha = (0xbf800000, -1) beta= (0x40000000, 2)
#### args: lda=4096 ldb=4096 ldc=4096
^^^ elapsed = 0.02335501 sec GFLOPS=5884.77
@@@ sgemm test OK
Testing dgemm
```

```
#### args: ta=0 tb=0 m=4096 n=4096 k=4096 alpha = (0x0000000000000000, 0) beta=
(0x0000000000000000, 0)
```

```
#### args: lda=4096 ldb=4096 ldc=4096
```

```
^^^ elapsed = 0.75175285 sec GFLOPS=182.825
```

```
@@@ dgemm test OK
```

```
==== Running N=10 without streams ====
```

```
Testing sgemm
```

```
#### args: ta=0 tb=0 m=4096 n=4096 k=4096 alpha = (0xbf800000, -1) beta= (0x00000000,
0)
```

```
#### args: lda=4096 ldb=4096 ldc=4096
```

```
^^^ elapsed = 0.23523593 sec GFLOPS=5842.6
```

```
@@@ sgemm test OK
```

```
Testing dgemm
```

```
#### args: ta=0 tb=0 m=4096 n=4096 k=4096 alpha = (0xbff0000000000000, -1) beta=
(0x0000000000000000, 0)
```

```
#### args: lda=4096 ldb=4096 ldc=4096
```

```
^^^ elapsed = 7.52091789 sec GFLOPS=182.742
```

```
@@@ dgemm test OK
```

```
==== Running N=10 with streams ====
```

```
Testing sgemm
```

```
#### args: ta=0 tb=0 m=4096 n=4096 k=4096 alpha = (0x40000000, 2) beta= (0x40000000, 2)
```

```
#### args: lda=4096 ldb=4096 ldc=4096
```

```
^^^ elapsed = 0.23394394 sec GFLOPS=5874.87
```

```
@@@ sgemm test OK
```

```
Testing dgemm
```

```
#### args: ta=0 tb=0 m=4096 n=4096 k=4096 alpha = (0xbff0000000000000, -1) beta=
(0x0000000000000000, 0)
```

```
#### args: lda=4096 ldb=4096 ldc=4096
```

```
^^^ elapsed = 7.50513196 sec GFLOPS=183.127
```

```
@@@ dgemm test OK
```

```
==== Running N=10 batched ====
```

```
Testing sgemm
```

```
#### args: ta=0 tb=0 m=4096 n=4096 k=4096 alpha = (0x3f800000, 1) beta= (0xbf800000, -1)
```

```
#### args: lda=4096 ldb=4096 ldc=4096
```

```

^^^^ elapsed = 0.59359002 sec GFLOPS=2315.39
@@@ sgemm test OK
Testing dgemm
#### args: ta=0 tb=0 m=4096 n=4096 k=4096 alpha = (0xbff0000000000000, -1) beta=
(0x4000000000000000, 2)
#### args: lda=4096 ldb=4096 ldc=4096
^^^^ elapsed = 8.48584008 sec GFLOPS=161.963
@@@ dgemm test OK

Test Summary
0 error(s)

```

如果运行结果如上所示，则说明 CUDA 环境安装正常。从上面例程我们也可以得到当前 GPU（这里是 Tesla M40，阿里云 HPC G4 实例）的计算能力，通过多种 Kernel 模式（单 Kernel、无 Stream、有 Stream、Batch 模式）测试了 SGEMM、DGEMM（即单/双精度矩阵乘）的计算性能。测试结论为双精度计算能力约为单精度的 1/32，与 NVIDIA 官方数据一致^[8]。

15.2.5 切换到 Caffe GPU 加速模式

Caffe 从 CPU 模式切换到 GPU 模式非常简单，首先修改 Makefile.config 中的选项：

```

# 仅 CPU 模式开关，这里要使用 GPU 模式，所以加上 “#”
# CPU_ONLY := 1

```

保存，退出。重新编译整个工程，命令为：

```

$ make clean
$ make -j

```

细心的读者会发现，这次编译调用了 nvcc 编译 Caffe 中的 CUDA 代码 (*.cu)。

而在 Windows 下需要修改 CommonSettings.props，修改其中三项内容如下：

```

<CpuOnlyBuild>false</CpuOnlyBuild>
<UseCuDNN>false</UseCuDNN>
<CuDnnPath></CuDnnPath>

```

等待编译成功。再次运行 MNIST 例程，运行前需要将训练超参数文件 examples/mnist/lenet_solver.prototxt 中最后一项 solver_mode 改为 GPU：

```

# solver mode: CPU or GPU
solver_mode: GPU

```

也可以在命令行显式加入选项“-gpu 0”，表示在 0 号 GPU 设备上运行 Caffe。在 GPU 加速模式下运行训练的效果与在 CPU 模式下没有区别，这里不再贴出。

下面仍然以 MNIST 例程中的 LeNet-5 模型为例，使用 `caffe time` 命令对 GPU 模式进行计时：

```
$ ./build/tools/caffe.bin time -model examples/mnist/lenet_train_test.prototxt -gpu 0
```

```
I0406 11:50:19.786551 19434 caffe.cpp:366] Average time per layer:
I0406 11:50:19.786571 19434 caffe.cpp:369]     mnist forward: 0.0517702 ms.
I0406 11:50:19.786605 19434 caffe.cpp:372]     mnist backward: 0.00244352 ms.
I0406 11:50:19.786623 19434 caffe.cpp:369]     conv1 forward: 2.86531 ms.
I0406 11:50:19.786644 19434 caffe.cpp:372]     conv1 backward: 7.57237 ms.
I0406 11:50:19.786669 19434 caffe.cpp:369]     pool1 forward: 0.0566336 ms.
I0406 11:50:19.786694 19434 caffe.cpp:372]     pool1 backward: 0.19289 ms.
I0406 11:50:19.786717 19434 caffe.cpp:369]     conv2 forward: 5.86639 ms.
I0406 11:50:19.786737 19434 caffe.cpp:372]     conv2 backward: 7.37989 ms.
I0406 11:50:19.786757 19434 caffe.cpp:369]     pool2 forward: 0.0279994 ms.
I0406 11:50:19.786777 19434 caffe.cpp:372]     pool2 backward: 0.073769 ms.
I0406 11:50:19.786798 19434 caffe.cpp:369]     ip1 forward: 0.163592 ms.
I0406 11:50:19.786819 19434 caffe.cpp:372]     ip1 backward: 0.186503 ms.
I0406 11:50:19.786839 19434 caffe.cpp:369]     relu1 forward: 0.0140166 ms.
I0406 11:50:19.786859 19434 caffe.cpp:372]     relu1 backward: 0.0140288 ms.
I0406 11:50:19.786878 19434 caffe.cpp:369]     ip2 forward: 0.114454 ms.
I0406 11:50:19.786898 19434 caffe.cpp:372]     ip2 backward: 0.0673651 ms.
I0406 11:50:19.786918 19434 caffe.cpp:369]     loss forward: 0.175297 ms.
I0406 11:50:19.786937 19434 caffe.cpp:372]     loss backward: 0.0324992 ms.
I0406 11:50:19.786972 19434 caffe.cpp:377] Average Forward pass: 9.44298 ms.
I0406 11:50:19.786990 19434 caffe.cpp:379] Average Backward pass: 15.626 ms.
I0406 11:50:19.787012 19434 caffe.cpp:381] Average Forward-Backward: 25.1487 ms.
I0406 11:50:19.787034 19434 caffe.cpp:383] Total Time: 1257.43 ms.
I0406 11:50:19.787050 19434 caffe.cpp:384] *** Benchmark ends ***
```

通过和 15.1 节 CPU 的结果对比，发现 GPU 模式相比 CPU 模式速度加快了 1 倍。由于 MNIST 模型较小，Caffe 实现的算法对 GPU 资源利用率不高，加速效果不明显。对于更大的模型（VGG-16 或 VGG-19），使用 GPU 模式将显著提升性能。

15.3 Caffe cuDNN 加速模式

为了达到更高的性能，我们可以借助专业加速库 cuDNN。本节我们会学习如何使用该加速

库提升 Caffe 计算速度。

15.3.1 获取 cuDNN

默认的 CUDA 安装包不包括 cuDNN，读者需要通过网站^[9]首先注册为 **CUDA 开发者** 才能获得下载权限，本文写作时最新版为 V5，读者可以选择与自己的 Caffe 版本、CUDA 版本对应的 cuDNN，推荐：CUDA 7.5 + cuDNN v3 + Caffe 20160303，这也是阿里云 HPC 上稳定的运行版本^[10]。如果读者执意尝新，可以参阅参考资料[11]了解如何填坑。图 15-6 给出了不同版本 cuDNN、不同 GPU 上的性能。

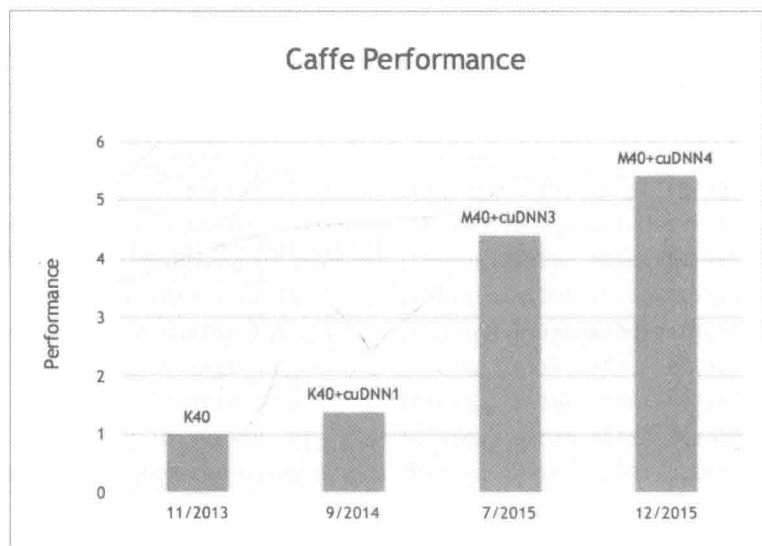


图15-6 历代cuDNN性能变化

15.3.2 切换到 Caffe cuDNN 加速模式

Caffe 从 GPU 模式切换到 cuDNN 模式非常简单，只需修改 Makefile.config 中的选项：

```
# cuDNN 加速开关，这里要使用 cuDNN，所以打开（去掉前面的“#”）该开关
USE_CUDNN := 1
```

保存，退出。重新编译整个工程，命令为：

```
$ make clean
$ make -j
```

在 Windows 下开启 cuDNN 加速选项, 需要修改 CommonSettings.props 中的三项内容如下:

```
<CpuOnlyBuild>false</CpuOnlyBuild>
<UseCuDNN>true</UseCuDNN>
<CuDnnPath>C:\Users\Administrator\Desktop\</CuDnnPath>
```

再次运行 MNIST 例程, 并使用 `caffe time` 命令计时:

```
$ ./build/tools/caffe.bin time -model examples/mnist/lenet_train_test.prototxt -gpu 0
```

```
I0406 11:47:29.436899 12103 caffe.cpp:366] Average time per layer:
I0406 11:47:29.436913 12103 caffe.cpp:369]     mnist forward: 0.0527168 ms.
I0406 11:47:29.436947 12103 caffe.cpp:372]     mnist backward: 0.00264768 ms.
I0406 11:47:29.436966 12103 caffe.cpp:369]     conv1 forward: 0.204704 ms.
I0406 11:47:29.436988 12103 caffe.cpp:372]     conv1 backward: 0.450452 ms.
I0406 11:47:29.437011 12103 caffe.cpp:369]     pool1 forward: 0.0553888 ms.
I0406 11:47:29.437033 12103 caffe.cpp:372]     pool1 backward: 0.192976 ms.
I0406 11:47:29.437062 12103 caffe.cpp:369]     conv2 forward: 0.348137 ms.
I0406 11:47:29.437088 12103 caffe.cpp:372]     conv2 backward: 0.884436 ms.
I0406 11:47:29.437104 12103 caffe.cpp:369]     pool2 forward: 0.0271654 ms.
I0406 11:47:29.437125 12103 caffe.cpp:372]     pool2 backward: 0.0740122 ms.
I0406 11:47:29.437149 12103 caffe.cpp:369]     ip1 forward: 0.167491 ms.
I0406 11:47:29.437170 12103 caffe.cpp:372]     ip1 backward: 0.188005 ms.
I0406 11:47:29.437191 12103 caffe.cpp:369]     relul forward: 0.024487 ms.
I0406 11:47:29.437212 12103 caffe.cpp:372]     relul backward: 0.0188403 ms.
I0406 11:47:29.437235 12103 caffe.cpp:369]     ip2 forward: 0.118394 ms.
I0406 11:47:29.437260 12103 caffe.cpp:372]     ip2 backward: 0.0677133 ms.
I0406 11:47:29.437281 12103 caffe.cpp:369]     loss forward: 0.146736 ms.
I0406 11:47:29.437302 12103 caffe.cpp:372]     loss backward: 0.0336544 ms.
I0406 11:47:29.437337 12103 caffe.cpp:377] Average Forward pass: 1.25336 ms.
I0406 11:47:29.437356 12103 caffe.cpp:379] Average Backward pass: 2.01538 ms.
I0406 11:47:29.437378 12103 caffe.cpp:381] Average Forward-Backward: 3.35128 ms.
I0406 11:47:29.437399 12103 caffe.cpp:383] Total Time: 167.564 ms.
I0406 11:47:29.437417 12103 caffe.cpp:384] *** Benchmark ends ***
```

从上面结果可以看出, Caffe cuDNN 模式相比 CPU 模式加速 15.46 倍, 相比 GPU 模式加速 7.7 倍, 效果惊人。

15.3.3 Caffe 不同硬件配置性能

前面一直使用 MNIST 例程，模型和数据规模都比较小，作为性能指标不够准确地反映硬件性能。这里我们测试 CaffeNet（Caffe 参考 ImageNet 模型）在不同 NVIDIA GPU 上的性能^[12]。

对于训练，每个时间点为 20 次迭代/每个迭代计算 256 张图片构成的 minibatch，总共 5120 张图片；对于测试，全部 50000 张验证集图片都会进行分类。

为了让性能最优，建议关掉 ECC，开启最大时钟速率。尽管 ECC 在速度上会引入可忽略的差异，但关掉它可以节省将近 1GB 的 GPU 存储器。

在关闭 ECC、最大时钟速率的最佳设置下，标准 Caffe 可以达到如下性能，如表 15-1 所示。

表 15-1 在关闭 ECC、最大时钟速率的最佳设置下标准 Caffe 性能

NVIDIA K40	Caffe	Caffe + cuDNN
训练	193.2 images/s	266.67 images/s
预测	500 images/s	823.72 images/s
NVIDIA K80	Caffe	Caffe + cuDNN
训练	193.2 images/s	266.67 images/s
预测	500 images/s	823.72 images/s
NVIDIA K20	Caffe	Caffe + cuDNN
训练	142.2 images/s	-
预测	376 images/s	-
NVIDIA Titan	Caffe	Caffe + cuDNN
训练	195 images/s	253 images/s
测试	500 images/s	754.1 images/s
NVIDIA GTX 770	Caffe	Caffe + cuDNN
训练	155.2 images/s	210.7 images/s
测试	387.6 images/s	480.7 images/s

下面介绍一下 K40 配置技巧。

为了让 K40 达到最高性能，需要关掉 ECC 并使能 boost 时钟速率（风险需要自己承担）。

关闭 ECC 的方法：

```
sudo nvidia-smi -i 0 --ecc-config=0 # 对每个 GPU 重复 -i x
```


然后重启。设置 GPU 模式为 persistence:

```
sudo nvidia-smi -pm 1
```

设置时钟速率:

```
sudo nvidia-smi -i 0 -ac 3004,875 # 对每个 GPU 重复 -i x
```

请注意这个设置, 每当驱动重新加载/重启时都会复位。在 Ubuntu 系统中将上述命令写入 `/etc/rc.local`。

15.4 练习题

1. 对比阅读 `conv_layer.cpp`、`conv_layer.cu`、`cudnn_conv_layer.cpp` 和 `cudnn_conv_layer.cu`, 体会同一算法的不同实现方式。
2. 查阅不同 GPU 架构 (Fermi、Kepler、Maxwell、Pascal) 的技术细节。
3. 为什么 Caffe 只需很少的工作量就能运行在 CPU、GPU、cuDNN 模式下? 如何设计类似的框架?
4. 如何在 Caffe 中增加新硬件平台 (例如 Xeon Phi、FPGA) 支持?

15.5 参考资料

- [1] NVIDIA CUDA C PROGRAMMING GUIDE v7.5
- [2] <https://developer.nvidia.com/cudnn>
- [3] <https://code.google.com/p/cuda-convnet2/>
- [4] maxDNN: An Efficient Convolution Kernel for Deep Learning with Maxwell GPUs, <http://arxiv.org/abs/1501.06633v3>
- [5] <http://docs.nvidia.com/cuda/index.html#axzz45RVcqwa8>
- [6] <http://blog.csdn.net/kkk584520/article/details/9413973>
- [7] CUDA Compiler Driver NVCC v7.5

[8] NVIDIA® TESLA® M40GPU ACCELERATOR Datasheet, <https://images.nvidia.com/content/tesla/pdf/nvidia-teslam40-datasheet.pdf>

[9] <https://developer.nvidia.com/cudnn>

[10] 阿里云 HPC 深度学习最佳实践, https://help.aliyun.com/document_detail/hpc/operation-reference/caffe.html

[11] Caffe + cuDNN v5, <http://blog.csdn.net/kkk584520/article/details/51163564>

[12] http://caffe.berkeleyvision.org/performance_hardware.html

第 16 天

Caffe 可视化方法

今天的内容是对第 14 天内容的进一步延伸，将一些工具需求具体化为可视化。可视化对于调参非常关键，一些网络设计灵感往往来源于可视化的结果，“艺术”与“科学”往往能在高维空间互通有无。

在很长一段时期内，学术界普遍认为神经网络中学习到的特征不好解释，而一些研究者则通过可视化方法理解卷积神经网络^[1]，针对上述质疑进行了有力回击。今天我们来欣赏一下这些由机器学到的作品。

16.1 数据可视化

我们前面使用 Caffe 进行图像分类，所有数据在预处理时都经历了从图像数据（二进制图像文件如 MNIST/CIFAR10、图片格式文件 JPEG/PNG）到 Caffe 数据库（LMDB/LEVELDB）的转换，这样做可以提高数据 I/O 速率，但代价是，开发者无法直观看到数据。

为了获得最佳体验，建议读者安装 Matlab R2014 以上版本软件（自行寻找资源，你们懂的）。

本节对数据可视化过程，不依赖 Caffe 环境，可以在任意位置运行代码。

16.1.1 MNIST 数据可视化

MNIST 数据可视化效果如图 16-1 所示。

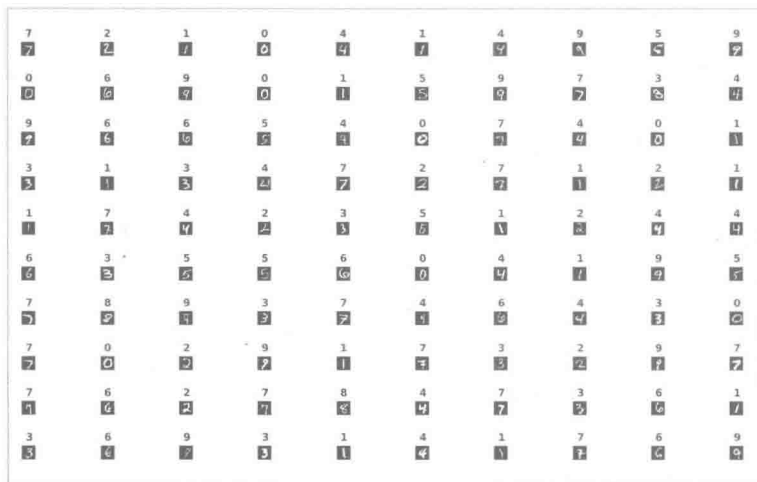


图16-1 MNIST数据可视化效果

Matlab 代码（show_mnist_data.m，可以放在\$CAFFE_ROOT/data/mnist/下，或者其他包含MNIST 数据文件的目录下运行）如下：

```
clear;
clc;
close all;

image_file_name = 't10k-images-idx3-ubyte';
index_file_name = 't10k-labels-idx1-ubyte';

fid1 = fopen(image_file_name,'rb');
fid2 = fopen(index_file_name,'rb');

images_data = fread(fid1,'uint8');
index_data = fread(fid2,'uint8');

fclose(fid1);
fclose(fid2);

images_data = images_data(17:end);
index_data = index_data(9:end);
image_buffer = zeros(28,28);

for k = 1 : 100: length(images_data)/28/28
```

```

figure(100);
for t = 1:100
    image_buffer = reshape(images_data((k + t - 2) * 28 * 28 + 1 : (k + t - 1) * 28 * 28), 28, 28);
    subplot(10, 10, t);
    imshow(uint8(image_buffer));
    title(num2str(index_data(k + t - 1)));
end
pause;
end

```

16.1.2 CIFAR10 数据可视化

CIFAR10 数据可视化效果如图 16-2 所示。

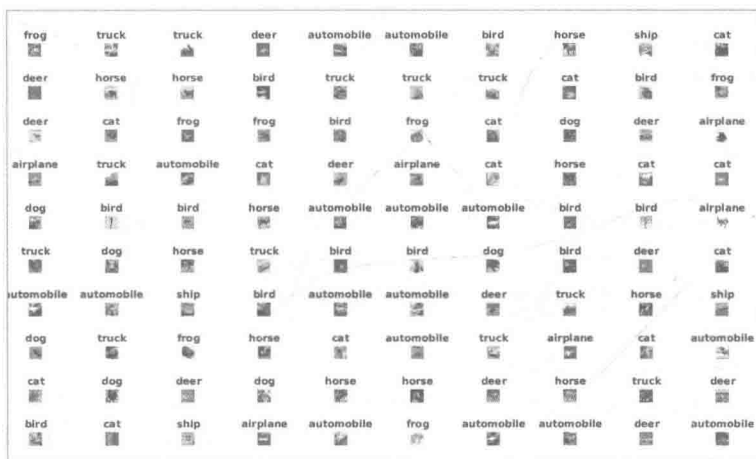


图16-2 CIFAR10 数据可视化效果

Matlab 代码 (show_cifar10_data.m, 可以放在\$CAFFE_ROOT/data/cifar10/下, 或者其他包含 CIFAR10 数据的目录下运行) 如下:

```

clear;
clc;
close all;
strings = {
    'airplane'
    'automobile'
    'bird'

```

```

    'cat'
    'deer'
    'dog'
    'frog'
    'horse'
    'ship'
    'truck'
};

image_file_name = 'data_batch_1.bin';
fid1 = fopen(image_file_name, 'rb');
images_data = fread(fid1, 'uint8');
fclose(fid1);

images_data = reshape(images_data, 3073, []);
image_idx = images_data(:, 1);

for k = 1 : 100 : size(images_data, 1)
    figure(100);
    for t = 1 : 100
        image_r = reshape(images_data(k + t - 1, 2 : 1025), 32, []);
        image_g = reshape(images_data(k + t - 1, 1026 : 2049), 32, []);
        image_b = reshape(images_data(k + t - 1, 2050 : 3073), 32, []);
        image_buffer = cat(3, image_r, image_g, image_b);
        subplot(10, 10, t);
        imshow(uint8(image_buffer));
        title(strings(image_idx(k + t - 1) + 1));
    end
    input('Press Enter to next picture :');
    pause;
end

```

16.1.3 ImageNet 数据可视化

ImageNet 数据集本身就是以图片形式提供的，利用操作系统自带的图片浏览器就可以显示。不过这样一张张看他（她或它）们的照片未免单调。斯坦福大学的博士生 Andrej Karpathy 使用 Caffe 提取 50000 张 ILSVRC 2012 验证集图像的 fc7 特征（4096 维向量），然后使用 Barnes-Hut t-SNE 得到与 L2 距离相关的二维嵌入图，如图 16-3 所示。



图16-3 利用t-SNE生成的二维嵌入图

在上面的二维嵌入图中，相似度高（二者 fc7 特征向量距离短）的两张图片会放置得比较近。

有兴趣的读者，可以结合 14.2 节使用 Caffe 做特征提取的内容并参考 t-SNE 文档^[2]尝试一下。

通过数据可视化，可以让设计者快速熟悉数据，并根据数据特点设计模型，有助于调试算法、提纯数据。数据是深度学习最重要的信息源，获取高质量的数据往往比设计复杂的模型更有效。

16.2 模型可视化

前面我们训练模型，只看到一堆运行日志，缺乏直观的可视化结果，到底“好”的模型长什么样子呢？

16.2.1 网络结构可视化

Caffe 提供了基于 Python 的网络结构可视化工具，需要编译 pycaffe 后使用。为此，我们借此机会学习 pycaffe 的编译和使用。为了尽量简短，本节只介绍 Ubuntu 14.04 下的安装方法，其他系统请根据实际情况调整。

1. 准备 Python 环境

```
$ sudo apt-get update
$ sudo apt-get install python-pip python-dev python-numpy
$ sudo apt-get install gfortran
$ sudo pip install -r ${CAFFE_ROOT}/python/requirements.txt
$ sudo pip install pydot
```




图16-7 GoogLeNet模型结构图

VGG-16, 模型结构如图 16-8 所示。



图16-8 VGG-16模型结构图

通过绘制网络架构我们可以直观地看到一个网络的整体框架到局部细节。在设计新模型结构时, 这种可视化工具能很快发现设计缺陷并加以改正, 而文本描述 (*.prototxt) 则不容易发现问题。

16.2.2 网络权值可视化

判断模型优劣的第二种方法是对训练后的模型权值进行可视化。

通常第一个卷积层是最容易解释的, 因为它直接“看”原始像素。其他更高层的滤波器权值也可以显示。训练过的 CaffeNet 第一个卷积层滤波器可视化效果如图 16-9 所示。

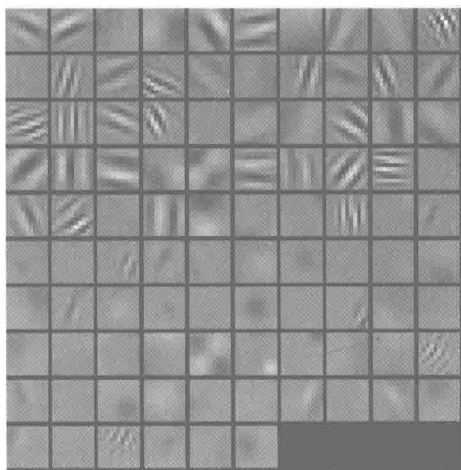


图16-9 CaffeNet Conv1 权值可视化效果

卷积层权值可视化十分有用, 因为经过良好训练的网络权值通常表现为美观、光滑的滤波器; 反之, 如果表现为噪声图样, 则可能意味着网络还没有经过足够长时间的训练, 或者由于正则化强度过小导致网络出现过拟合。从图 16-9 可以看出, CaffeNet Conv1 权值非常美观、平滑, 说明网络收敛效果不错。另外, 也可以观察到一部分权值负责提取高频灰度特征; 而另一

部分负责提取低频彩色特征。

用于显示图 16-9 的 Matlab 代码 (convl_weights_vis.m, 放在 Caffe 根目录下, 需要预先编译 matcaffe) 如下:

```
clear;
clc;
close all;
addpath('matlab')
caffe.set_mode_cpu();
caffe.version()

net = caffe.Net('models/bvlc_reference_caffenet/deploy.prototxt', 'models/bvlc_
reference_caffenet/bvlc_reference_caffenet.caffemodel', 'test');
net.layer_names
net.blob_names
convl_layer = net.layer_vec(2);
blob1 = convl_layer.params(1);
w = blob1.get_data();
size(w)
W = zeros(11 * 3, 11 * 96);
for u = 1:3
    for v = 1:96
        W(11 * (u - 1) + (1:11), 11 * (v - 1) + (1:11)) = w(:, :, u, v)';
    end
end

W = W - min(min(W));
W = W / (max(max(W))) * 255;
W = uint8(W);
W = [W, zeros(size(W, 1), 4 * 11)];
WW = cat(3, W(1:11, :), W(12:22, :), W(23:33, :));
W = zeros(10 * 12, 10 * 12, 3);
for u = 1:10
    for v = 1:10
        W((u-1)*12 + (1:11), (v - 1) * 12 + (1:11), :) = WW(:, (u-1) * 11 * 10 + (v - 1)
* 11 + (1:11), :);
    end
end

W = uint8(W);
figure; imshow(W);
```

编译 `matcaffe` 比较简单，只需在安装 Matlab 成功后，将 Matlab 目录更新至 Caffe 的 `Makefile.config`:

```
# This is required only if you will compile the matlab interface.
# MATLAB directory should contain the mex binary in /bin.
MATLAB_DIR := /Your/Path/To/MATLAB/R2015b/
# MATLAB_DIR := /Applications/MATLAB_R2012b.app
```

保存，退出，然后编译：

```
make matcaffe
```

等待编译结束即可。如果编译报错，请核对 Matlab 路径。

我们需要从 Caffe Model Zoo 获取已经训练好的 CaffeNet 权值文件，这样就不需要从头训练一个 CaffeNet 模型了，节省了时间和计算资源。获取方式如下：

```
$ cd models/bvlc_reference_caffenet/
$ wget http://dl.caffe.berkeleyvision.org/bvlc_reference_caffenet.caffemodel
```

CaffeNet 更高层的权值可视化效果如图 16-10 至图 16-13 所示。

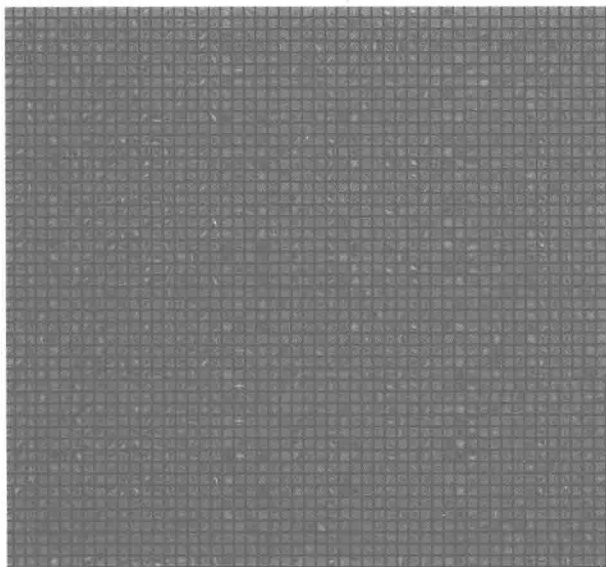


图16-10 CaffeNet Conv2权值可视化效果（部分）

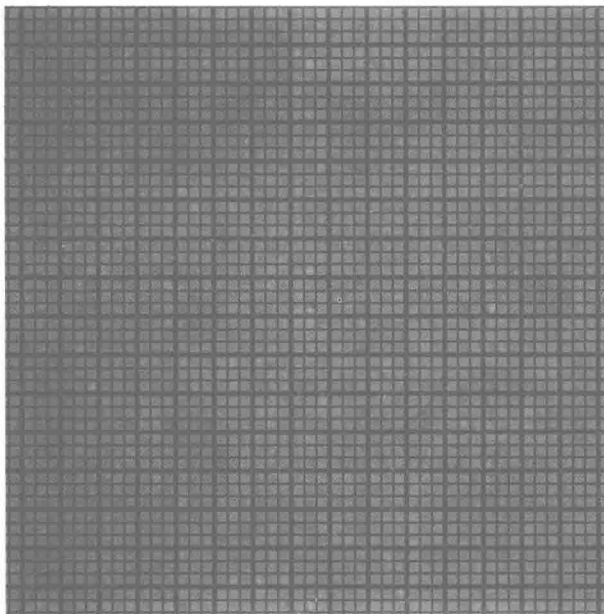


图16-11 CaffeNet Conv3权值可视化效果（部分）

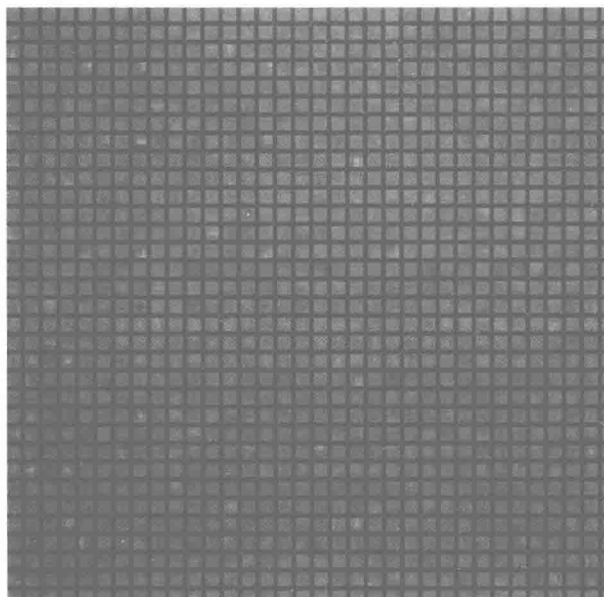


图16-12 CaffeNet Conv4权值可视化效果（部分）

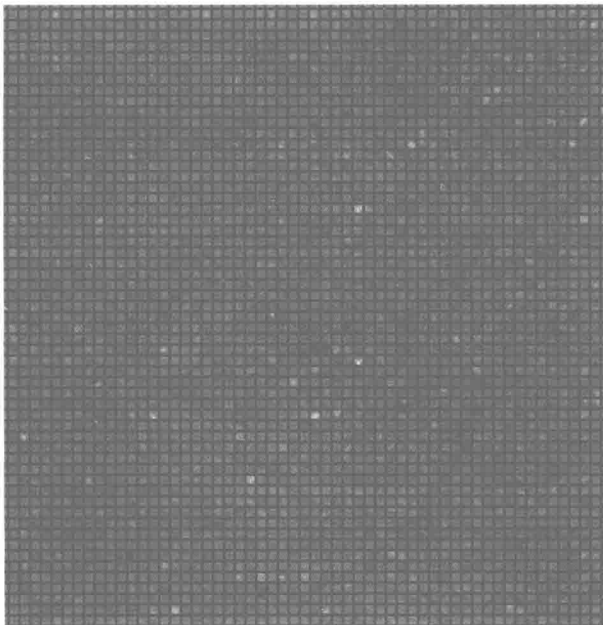


图16-13 CaffeNet Conv5权值可视化效果（部分）

从图 16-10 可以看到，CaffeNet 第二个卷积层可视化效果虽然不像第一层那样具有较强的可解释性，但显然仍然光滑、形状规则、无噪声图样。更高层（Conv3、Conv4、Conv5）区别不大，可读性更差了。

Conv2~Conv5 权值可视化的 Matlab 代码（visualize_weights.m，放于 Caffe 根目录下）如下：

```
function [] = visualize_weights(w, s)
h = max(size(w, 1), size(w, 2)); % Kernel size
g = h + s; % Grid size, larger than Kernel size for better visual effects.

% Normalization for gray scale
w = w - min(min(min(min(w))));
w = w / max(max(max(max(w)))) * 255;
w = uint8(w);

W = zeros(g * size(w, 3), g * size(w, 4));
for u = 1:size(w, 3)
    for v = 1:size(w, 4)
        W(g * (u - 1) + (1:h), g * (v - 1) + (1:h)) = w(:, :, u, v)';
    end
end
```

```
end
W = uint8(W);
figure;imshow(W);
```

Matlab 主函数代码 (caffenet_weights_vis.m, 放于 Caffe 根目录下) 如下:

```
clear;
clc;
close all;
addpath('matlab')
caffe.set_mode_cpu();
fprintf(['Caffe Version = ', caffe.version(), '\n']);

net = caffe.Net('models/bvlc_reference_caffenet/deploy.prototxt', 'models/bvlc_
reference_caffenet/bvlc_reference_caffenet.caffemodel', 'test');

fprintf('Load net done. Net layers : ');
net.layer_names

fprintf('Net blobs : ');
net.blob_names

% Conv1 Weight Visualization
conv1_layer = net.layer_vec(2);
blob1 = conv1_layer.params(1);
w = blob1.get_data();
fprintf('Conv1 Weight shape: ');
size(w)
visualize_weights(w, 1);

% Conv2 Weight Visualization
conv2_layer = net.layer_vec(6);
blob2 = conv2_layer.params(1);
w2 = blob2.get_data();
fprintf('Conv2 Weight shape: ');
size(w2)
visualize_weights(w2, 1);

% Conv3 Weight Visualization
conv3_layer = net.layer_vec(10);
blob3 = conv3_layer.params(1);
```

```
w3 = blob3.get_data();
fprintf('Conv3 Weight shape: ');
size(w3)
visualize_weights(w3, 1);

% Conv4 Weight Visualization
conv4_layer = net.layer_vec(12);
blob4 = conv4_layer.params(1);
w4 = blob4.get_data();
fprintf('Conv4 Weight shape: ');
size(w4)
visualize_weights(w4, 1);

% Conv5 Weight Visualization
conv5_layer = net.layer_vec(14);
blob5 = conv5_layer.params(1);
w5 = blob5.get_data();
fprintf('Conv5 Weight shape: ');
size(w5)
visualize_weights(w5, 1);
```

下面列举一些反例——什么样的权值是“坏”的？见图 16-14、图 16-15、图 16-16。

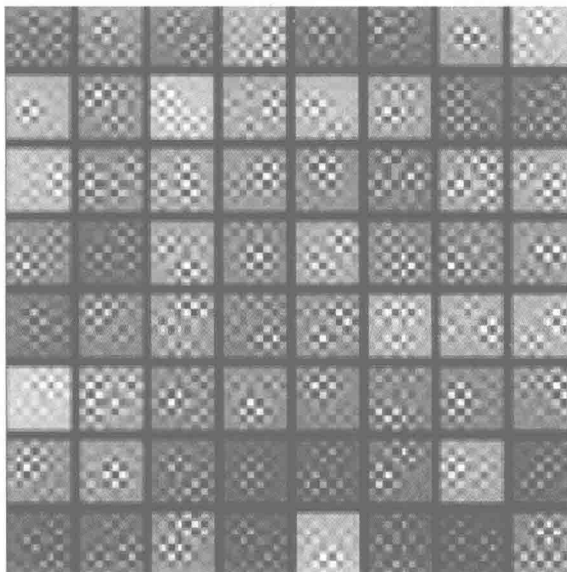


图16-14 图案类似噪声

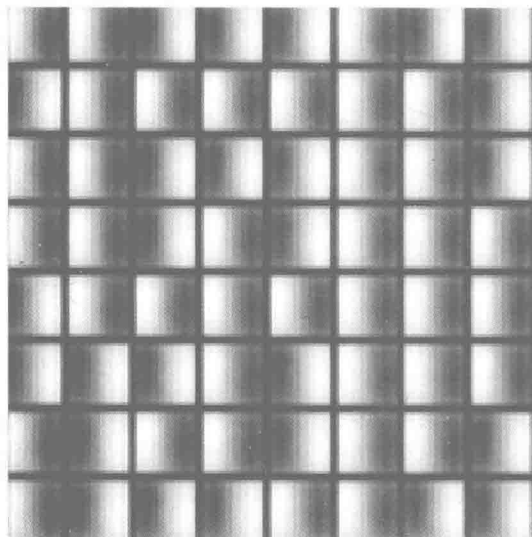


图16-15 图案相关性太高

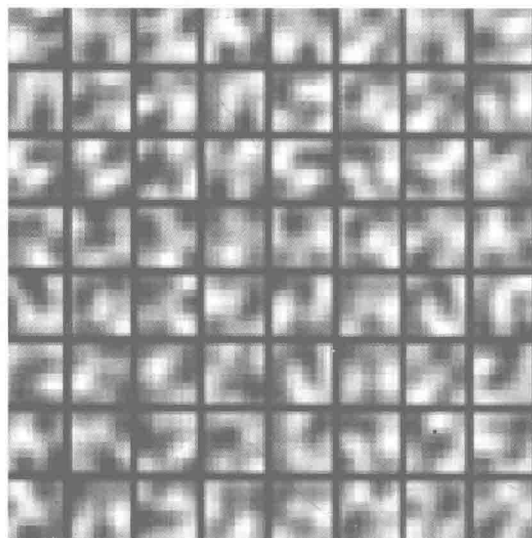


图16-16 图案缺乏结构性

16.3 特征图可视化

我们在前两节阅尽了数据和模型，还有一类数据没有留意，即网络在前向传播阶段的各层响应。下面我们对每层响应进行可视化。

使用可视化可以很容易看出危险的错误信号，比如对于多个不同的输入，一些响应特征图全为零，这样可以检测“死”滤波器，可能是学习速率过高的症状。

使用一个训练过的 CaffeNet 模型（仍使用 `models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel`）对一张猫的图片（这里使用 `examples/images/cat.jpg`，见图 16-17）进行分类，我们利用 `matcaffe` 提供的接口可以打印出模型各层对输入图像的响应，即特征图。



图16-17 可爱的小猫作为输入数据

CaffeNet 模型对输入图像产生的响应如图 16-18 至图 16-23 所示。

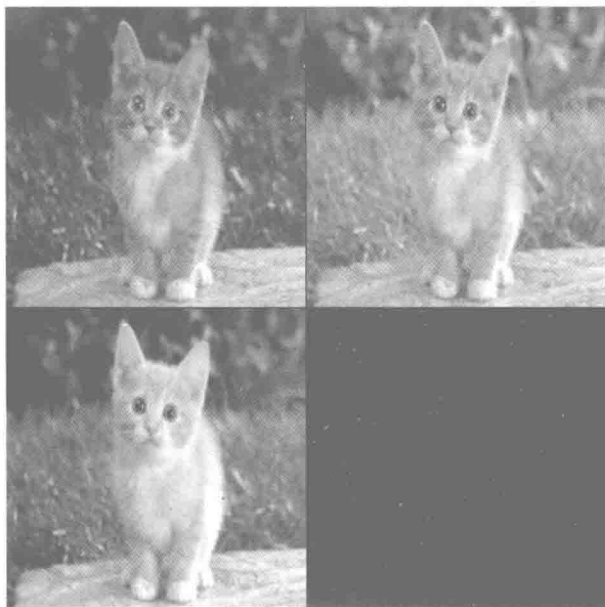


图16-18 数据层输出



图16-19 第一个卷积层的响应



图16-20 第二个卷积层的响应特征图

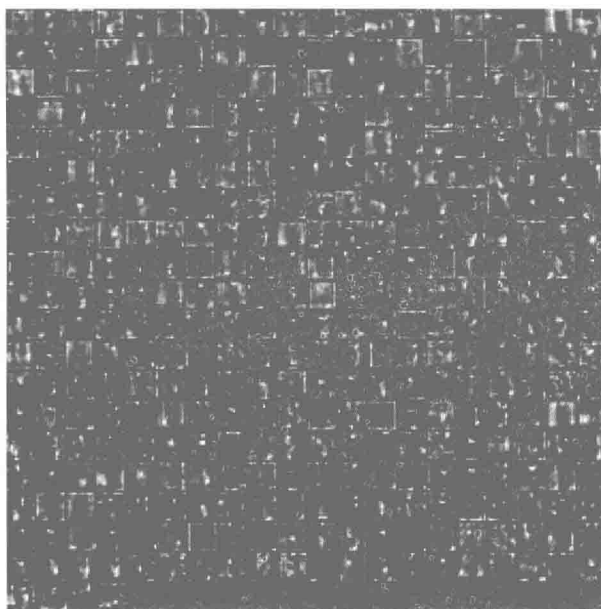


图16-21 第三个卷积层的响应

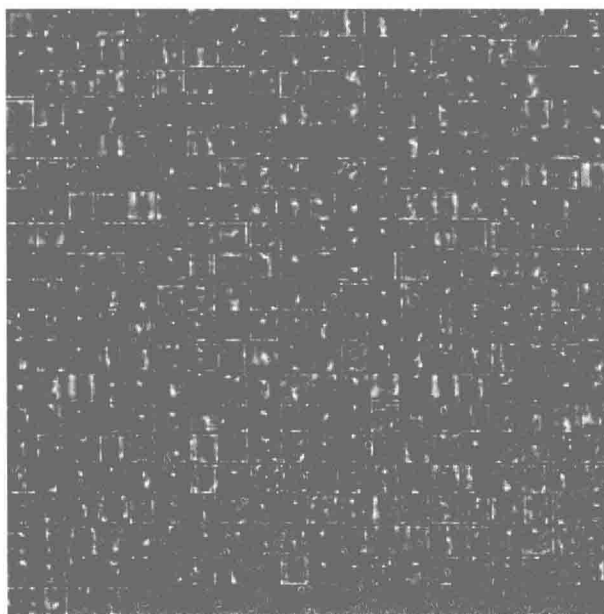


图16-22 第四个卷积层的响应

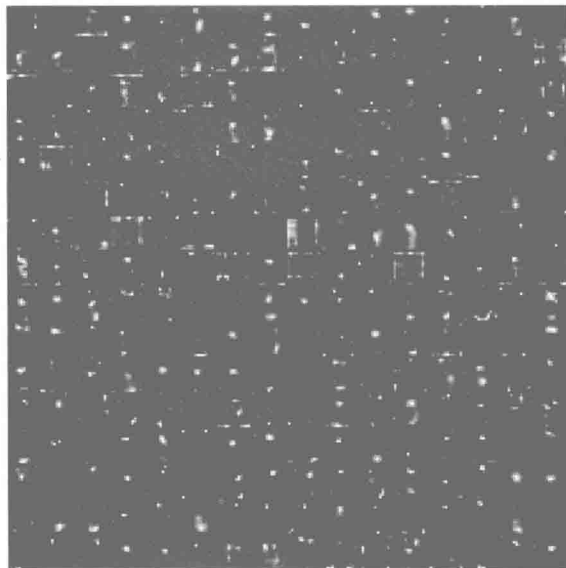


图16-23 第五个卷积层的典型响应

每个小方块显示了对应特定滤波器的响应图。注意到低层响应特征图关注图像中不同细节（背景或主体的纹理或轮廓），我们从中可以选择图片风格特征（详见 19.6 节内容），高维响应特征图变得局部且稀疏，用于剔除不相关内容并提取目标重要的特征。

本节 Matlab 代码（visualize_feature_maps.m，放在 Caffe 根目录下，需要预先编译 matcaffe）如下：

```
function [] = visualize_feature_maps(w, s)
h = max(size(w, 1), size(w, 2)); % Feature map size
g = h + s;
c = size(w, 3);
cv = ceil(sqrt(c));
W = zeros(g * cv, g * cv);

for u = 1:cv
    for v = 1:cv
        tw = zeros(h, h);
        if((u - 1) * cv + v) <= c
            tw = w(:, :, (u - 1) * cv + v, 1)';
            tw = tw - min(min(tw));
            tw = tw / max(max(tw)) * 255;
        end
    end
end
```

```

W(g * (u - 1) + (1:h), g * (v - 1) + (1:h)) = tw;
end
end
W = uint8(W);
figure;imshow(W);

```

Matlab 主函数代码 (fm_visual.m, 放在 Caffe 根目录下) 如下:

```

clear;
clc;
close all;
addpath('matlab')
caffe.set_mode_cpu();
fprintf(['Caffe Version = ', caffe.version(), '\n']);

net = caffe.Net('models/bvlc_reference_caffenet/deploy.prototxt',
'models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel', 'test');

fprintf('Load net done. Net layers : ');
net.layer_names

fprintf('Net blobs : ');
net.blob_names

fprintf('Now preparing data...\n');
im = imread('examples/images/cat.jpg');
figure;imshow(im);title('Original Image');
d = load('matlab/+caffe/imagenet/ilsvrc_2012_mean.mat');
mean_data = d.mean_data;
IMAGE_DIM = 256;
CROPPED_DIM = 227;

% Convert an image returned by Matlab's imread to im_data in caffe's data
% format: W x H x C with BGR channels
im_data = im(:, :, [3, 2, 1]); % permute channels from RGB to BGR
im_data = permute(im_data, [2, 1, 3]); % flip width and height
im_data = single(im_data); % convert from uint8 to single
im_data = imresize(im_data, [IMAGE_DIM IMAGE_DIM], 'bilinear'); % resize im_data

```

```
im_data = im_data - mean_data; % subtract mean_data (already in W x H x C, BGR)
im = imresize(im_data, [CROPPED_DIM CROPPED_DIM], 'bilinear'); % resize im_data
km = cat(4, im, im, im, im, im);
pm = cat(4, km, km);
input_data = {pm};

scores = net.forward(input_data);

scores = scores{1};
scores = mean(scores, 2); % take average scores over 10 crops

[~, maxlabel] = max(scores);

maxlabel

figure; plot(scores);

fm_data = net.blob_vec(1);
d1 = fm_data.get_data();
fprintf('Data size = ');
size(d1)
visualize_feature_maps(d1, 1);

fm_conv1 = net.blob_vec(2);
f1 = fm_conv1.get_data();
fprintf('Feature map conv1 size = ');
size(f1)
visualize_feature_maps(f1, 1);

fm_conv2 = net.blob_vec(5);
f2 = fm_conv2.get_data();
fprintf('Feature map conv2 size = ');
size(f2)
visualize_feature_maps(f2, 1);

fm_conv3 = net.blob_vec(8);
f3 = fm_conv3.get_data();
fprintf('Feature map conv3 size = ');
size(f3)
```

```

visualize_feature_maps(f3, 1);

fm_conv4 = net.blob_vec(9);
f4 = fm_conv4.get_data();
fprintf('Feature map conv4 size = ');
size(f4)
visualize_feature_maps(f4, 1);

fm_conv5 = net.blob_vec(10);
f5 = fm_conv5.get_data();
fprintf('Feature map conv5 size = ');
size(f5)
visualize_feature_maps(f5, 1);

```

通过观察每层的响应特征图，我们可以判断模型的结构设计（如每层通道数目）是否合理，如果大量的响应特征图都重复出现或全为接近 0 的值，则可以减少通道数目以提高网络效率。

16.4 学习曲线

我们可以利用 Caffe 运行 log 绘制训练过程中的学习曲线。

首先要把 Caffe 运行产生的所有 log 重定向到文件，使用如下语句：

```
$ ./examples/cifar/train_quick.sh >& cifar.log &
```

其中，“>&”表示所有的标准输出（stdout）和标准错误输出（stderr）都将被重定向，“cifar.log”为重定向后 log 保存的文件，最后的“&”表示将命令放入后台执行。为了观察是否正常运行，可以在程序运行阶段，使用“tail -f cifar.log”连续观测 log 文件的更新，退出使用“Ctrl + C”。

TIPS：将训练任务放入后台有什么好处？

第一，你可以用一个终端干更多的事情，比如一边跑训练，一边写代码；

第二，当你的终端（有意或无意）关闭时，不会停止训练进程；

第三，你可以随时随地从其他终端（远程）查看训练进度，而不必要回到启动训练任务的那个终端。

使用如下 Shell 命令提取 log 文件中的 loss 值：

```
$ cat cifar.log | grep "Train net output" | awk '{print $11}'
```

其中，“|”为管道命令，即将前一条命令的输出直接送入后一条命令作为输入。

```
2.30241
1.74617
1.65306
1.30551
1.20516
1.25205
1.18319
1.19186
0.922211
0.937064
1.02281
0.961253
0.928429
0.699045
0.822131
0.840853
0.811154
0.827273
0.65812
0.759129
0.807019
.....
```

这些值即为训练时 loss 值，为了打印出曲线，我们使用 Matlab 工具结合命令行提取日志中的 loss 数值，并绘制曲线，如图 16-24 所示。

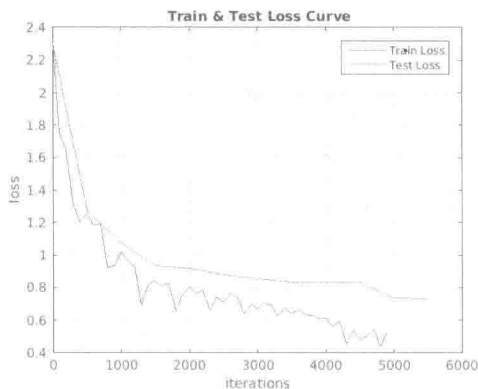


图16-24 CIFAR10学习曲线

通过学习曲线，可以评估当前训练状态：

- train loss 不断下降，test loss 不断下降，说明网络仍然在认真学习。
- train loss 不断下降，test loss 趋于不变，说明网络过拟合。
- train loss 趋于不变，test loss 趋于不变，说明学习遇到瓶颈，需减小学习速率或批量数据尺寸。
- train loss 趋于不变，test loss 不断下降，说明数据集 100%有问题。
- train loss 不断上升，test loss 不断上升（最终变为 NaN），可能是网络结构设计不当、训练超参数设置不当、程序 bug 等某个问题引起的，需要进一步定位。

本节 Matlab 代码（show_loss_curve.m，放在 Caffe 根目录下）如下：

```
clear;
clc;
close all;
% 这个参数用来指定 Caffe 运行 log 文件
% 生成 log 文件方法：./examples/cifar10/train_quick.sh >& cifar.log&
train_log_file = 'cifar.log';
% 这个参数相当于 solver.prototxt 中的 display 值
train_interval = 100;
% 这个参数相当于 solver.prototxt 中的 test_interval 值
test_interval = 500;

[~, string_output] = dos(['cat ', train_log_file, ' | grep ''Train net output #0'' | awk ''{print $1}''']);
% fid = fopen('matlab_train_loss', 'r');
% train_loss = fscanf(fid, '%f\n');
% fclose(fid);
train_loss = str2num(string_output);
n = 1:length(train_loss);
idx_train = (n - 1) * train_interval;

% fid = fopen('matlab_test_loss', 'r');
% test_loss = fscanf(fid, '%f\n');
% fclose(fid);
[~, string_output] = dos(['cat ', train_log_file, ' | grep ''Test net output #1'' | awk
```

```

''{print $1}''});
test_loss = str2num(string_output);
m = 1:length(test_loss);
idx_test = (m - 1) * test_interval;
figure;plot(idx_train, train_loss);
hold on;
plot(idx_test, test_loss);

grid on;
legend('Train Loss', 'Test Loss');
xlabel('iterations');
ylabel('loss');
title(' Train & Test Loss Curve');

```

16.5 小结

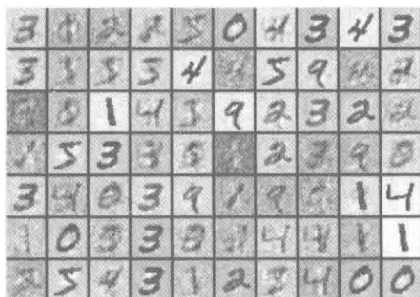
俗话说，耳听为虚，眼见为实。今天主要学习了如何“看”Caffe 中的细节信息，包括数据、模型、特征图和训练时的学习曲线。多“看”能获得很多有益的提示，指导我们更好地设计模型，调试程序^{[3][4][5]}。“看破”Caffe，释然于心。

16.6 练习题

1. 学习 Python 或 Matlab 可视化编程，并使用代码生成下图（数据来源：CIFAR10）：



2. 使用你所掌握的可视化技术，观察 MNIST 分类过程中的特征图，练习生成下图：



3. 如何判断模型过拟合？如何判断模型欠拟合？分别应采取哪些措施？

16.7 参考资料

[1] Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, arXiv:1312.6034v2

[2] User's Guide for t-SNE Software

[3] Understanding Deep Image Representations by Inverting Them, arXiv:1412.0035v1

[4] EXPLAINING AND HARNESSING ADVERSARIAL EXAMPLES, arXiv:1412.6572v3

[5] Do Convnets Learn Correspondence?

第 17 天

Caffe 迁移和部署

今天我们关注 Caffe 在不同平台上迁移（又称为移植）部署，这是实际工作中必不可少的内容，无论用 Caffe 做互联网在线业务（在线图像识别、文字识别、人脸识别）还是嵌入式系统（智能机器人、自动驾驶系统），都会用到今天将要介绍的内容。

17.1 从开发测试到生产部署

业余 DL 爱好者受实际软硬件条件限制，一般不会区分“开发测试”和“生产部署”这两个环境，所有需要的软件包都安装在同一台机器上。而一旦作为产品对外提供服务时，就会有很大问题。一方面，开发测试机器的环境具有相当大的不确定性，例如编译新版本 Caffe 覆盖旧版本，一些模块可能需要做适配才能正常工作，需要停服维护；另一方面，当业务量不断扩大时，生产环境所需的节点数目与日俱增，不可能在每个节点上分别建立开发环境，而应使用批量部署（克隆）技术，将打包后的软件和所需的运行环境自动化推送到所有生产节点。

深度学习是新兴的业务类型，相比 Web Server、数据库这类传统业务具有很多不同点。首先，深度学习依赖高性能计算服务器，而传统业务只需很少的计算量；其次，深度学习业务一般对延迟不敏感。

深度学习业务从开发测试到生产部署可以用一句话描述：离线训练、在线识别。为什么不采用在线训练？主要出于两方面考虑：一方面，在线服务器占用公网带宽资源，而训练相当消耗计算能力，对网络带宽需求不高，造成资源浪费；另一方面，嵌入式系统本身计算能力较弱，为保证电池续航时间不适合运行训练任务。

如图 17-1 所示，完整的深度学习开发周期从逻辑上分为开发和部署两个阶段。在开发阶段

(离线训练阶段), 主要由数据专家来选择训练数据, 由算法专家设计模型参数, 由开发专家对训练过程进行优化和调试, 得到满足发布的模型, 在 Caffe 中即为*.caffemodel 文件。在部署阶段(在线识别阶段), 由线上负责生产的工程师利用开发团队提供的可发布模型部署到线上生产机器, 接入线上其他服务如存储、数据库, 获取在线数据(可能直接来自客户)并使用上述模型处理, 将得到的结果(分类/检测/分割)返回客户端或指定的存放结果的文件服务器。生产阶段的一些异常结果(错误分类)也可反馈给开发阶段, 指导数据专家剔除“脏”数据、算法专家改进模型、开发专家排查代码 bug, 得到更准确的模型。

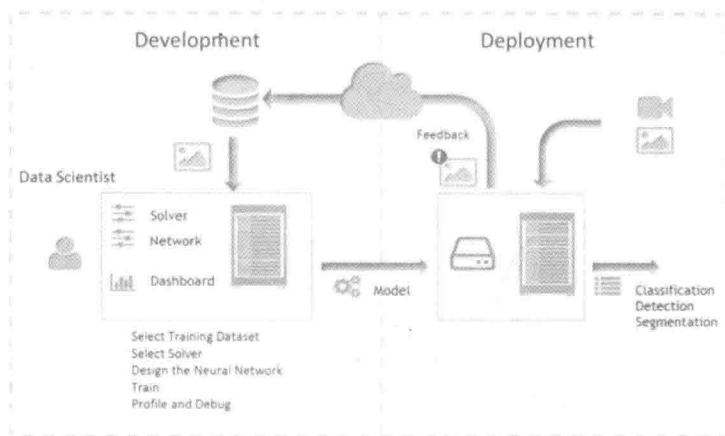


图17-1 深度学习开发周期

生产机器除了上述互联网线上服务器之外, 也可能是某个嵌入式平台, 如图 17-2 所示的 Jetson TX1^[1]。

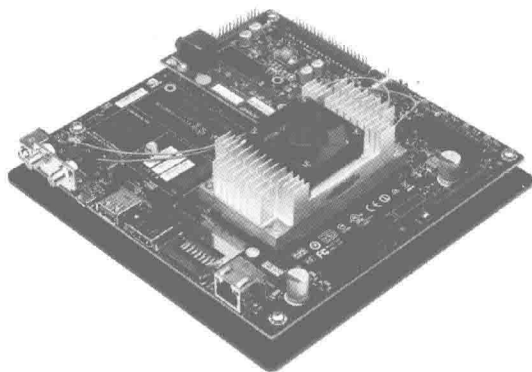


图17-2 Jetson TX1

Jetson TX1 是一个嵌入式 CPU + GPU 平台，CPU 为 64 位 ARM A57，GPU 为 NVIDIA Maxwell 架构，拥有 256 个 CUDA 核心和 4GB LPDDR4 内存，运算能力达 1 TFLOPS。在嵌入式设备中，I/O 模块必不可少，除了传统的 UART、I2C、I2S、SPI 接口外，Jetson TX1 还具有 4K 编/解码模块、DP/HDMI 视频输出接口和每秒接收 14 亿像素的高速相机接口，即每秒可以接收大约 117 张 iPhone 6s 拍摄的照片（iPhone 6s 相机为 1200 万像素）。这些特性使得该平台非常适合嵌入式计算机视觉应用，如智能机器人或自动驾驶汽车。

使用 Jetson TX1 部署 Caffe 模型非常简单，无须修改代码，只需将开发阶段得到的 *.caffemodel 拷贝至 SD 卡，插入开发板并设置合理的运行参数，机器即可迅速获得相应智能。

本节从宏观上介绍了从开发测试到生产部署的过程，下一节将介绍具体的实现方法。

17.2 使用 Docker

Docker^[2]是一个开源的应用容器引擎，开发者可以把一个 Linux 应用和它所依赖的一切（比如配置文件和库）都封装到一个容器中，然后发布到任何 Linux 机器上。“容器”技术是一种沙箱机制，不同实例之间互不影响（类似于 iPhone 的 app）。容器与虚拟机不同，不需要运行操作系统，而是共享主机上的操作系统，所以几乎没有性能开销，可以很容易地在机器和数据中心中运行。

Docker 作为一种全新的自动化运维工具，可以实现开发测试环境与生产部署环境的平滑迁移，大大提高了开发上线的效率。目前已有大量互联网公司使用 Docker 技术取代传统的生产部署发布流程。

17.2.1 Docker 基本概念

首先我们需要理解 Docker 最重要的三个概念：镜像（Image）、容器（Container）、镜像仓库（Docker Hub）。

- 镜像（Image）：一个包含了应用程序和其运行时依赖环境的只读文件（可类比为系统盘、可执行程序文件），它是构建容器的模板，通过一个镜像我们可以构造出很多相互独立但运行环境一样的容器。
- 容器（Container）：基于某个镜像生成并动态运行的相互隔离的实例（可类比为运行起来的操作系统、运行可执行程序文件的进程）。

- 镜像仓库 (Docker Hub): Docker 官方提供的用于集中存储、管理镜像的服务, 采用类似于 Git Hub 的方式保存公有或私有的镜像, 同时允许第三方搭建。

如图 17-3 所示就是典型的 Docker 工作流程, 通过此图能清晰地理解这三个重要概念之间的关系。

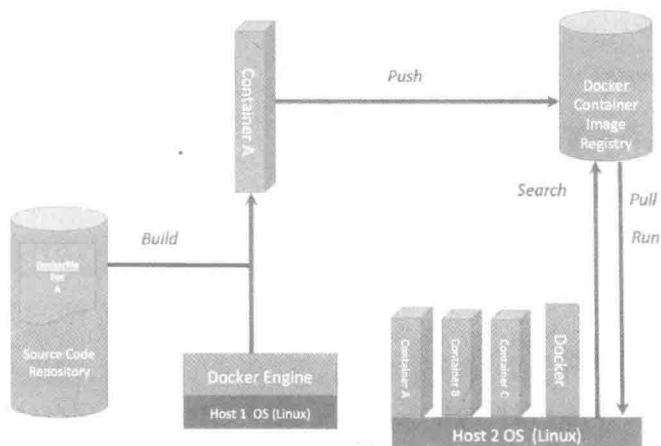


图17-3 Docker工作流程

图 17-3 中, 包括两个主机 Host 1 和 Host 2, 其中 Host 1 用来构建 Docker 镜像, 对应于上节的开发测试机器, 由开发工程师定制镜像中的内容并打包; Host 2 对应于上节的生产部署机器, 由生产工程师获取指定镜像并运行。开发和生产机器之间的镜像通过镜像仓库统一管理和分发。接下来我们一步步学习这些操作流程。

17.2.2 Docker 安装

目前支持 Docker 的操作系统有 Ubuntu 14.04、Cent OS 7、Fedora 21 等。Docker 安装过程需要使用 root 权限。

1. Ubuntu 安装步骤

```
# apt-get update
# apt-get -y install docker.io
```

检验 Docker 服务的状态:

```
# service docker.io status
docker.io start/running, process 14394
```

把 Docker 安装为自启动服务，让它随服务器的启动而自动运行，执行命令：

```
# update-rc.d docker.io defaults
```

2. Cent OS 7、Fedora 21 安装步骤

```
# yum install -y docker
```

启动 Docker 服务：

```
# systemctl start docker
```

检查服务状态是否正常：

```
# systemctl status -l docker
```

设置开机自启动：

```
# systemctl enable docker
```

3. 添加 Docker 用户组

在默认情况下，只有 root 用户具有 Docker 使用权限。这样不利于团队协作。我们可以创建一个 docker 用户组，将所有需要使用 Docker 服务的用户添加到用户组中，尽量避免多人同时使用 root 用户。

```
# usermod -aG docker your_user_name
```

现在可以退出 root 用户，使用 docker 用户组中的用户运行 Docker 命令了。

4. 测试 Docker 安装成功

运行一个 hello world 例程，如果正常则说明前面配置没问题。正常的输出为：

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from hello-world
d59cd4c39e50: Pull complete
f1d956dc5945: Pull complete
Digest: sha256:4f32210e234b4ad5cac92efacc0a3d602b02476c754f13d517e1ada048e5a8ba
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
```


1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker Hub account:

<https://hub.docker.com>

For more examples and ideas, visit:

<https://docs.docker.com/engine/userguide/>

如果一切正常，恭喜，你代表了先进生产力的发展方向。继续前进！

17.2.3 Docker 入门

在安装了 Docker 的系统命令行中输入 docker 并运行，可获得如下命令详单：

```
$ docker
```

```
Usage: docker [OPTIONS] COMMAND [arg...]
```

```
A self-sufficient runtime for linux containers.
```

```
Options:
```

<code>--api-cors-header=</code>	Set CORS headers in the remote API
<code>-b, --bridge=</code>	Attach containers to a network bridge
<code>--bip=</code>	Specify network bridge IP
<code>-D, --debug=false</code>	Enable debug mode
<code>-d, --daemon=false</code>	Enable daemon mode
<code>--default-ulimit=[]</code>	Set default ulimits for containers
<code>--dns=[]</code>	DNS server to use
<code>--dns-search=[]</code>	DNS search domains to use
<code>-e, --exec-driver=native</code>	Exec driver to use
<code>--fixed-cidr=</code>	IPv4 subnet for fixed IPs
<code>--fixed-cidr-v6=</code>	IPv6 subnet for fixed IPs
<code>-G, --group=docker</code>	Group for the unix socket

```

-g, --graph=/var/lib/docker      Root of the Docker runtime
-H, --host=[]                    Daemon socket(s) to connect to
-h, --help=false                 Print usage
--icc=true                      Enable inter-container communication
--insecure-registry=[]          Enable insecure registry communication
--ip=0.0.0.0                    Default IP when binding container ports
--ip-forward=true               Enable net.ipv4.ip_forward
--ip-masq=true                  Enable IP masquerading
--iptables=true                Enable addition of iptables rules
--ipv6=false                   Enable IPv6 networking
-l, --log-level=info            Set the logging level
--label=[]                     Set key=value labels to the daemon
--log-driver=json-file          Containers logging driver
--mtu=0                         Set the containers network MTU
-p, --pidfile=/var/run/docker.pid Path to use for daemon PID file
--registry-mirror=[]           Preferred Docker registry mirror
-s, --storage-driver=           Storage driver to use
--selinux-enabled=false        Enable selinux support
--storage-opt=[]               Set storage driver options
--tls=false                    Use TLS; implied by --tlsverify
--tlscacert=~/.docker/ca.pem   Trust certs signed only by this CA
--tlscert=~/.docker/cert.pem   Path to TLS certificate file
--tlskey=~/.docker/key.pem     Path to TLS key file
--tlsverify=false              Use TLS and verify the remote
-v, --version=false            Print version information and quit

```

Commands:

```

attach  Attach to a running container
build   Build an image from a Dockerfile
commit  Create a new image from a container's changes
cp      Copy files/folders from a container's filesystem to the host path
create  Create a new container
diff    Inspect changes on a container's filesystem
events  Get real time events from the server
exec    Run a command in a running container
export  Stream the contents of a container as a tar archive
history Show the history of an image
images  List images
import  Create a new filesystem image from the contents of a tarball

```

info	Display system-wide information
inspect	Return low-level information on a container or image
kill	Kill a running container
load	Load an image from a tar archive
login	Register or log in to a Docker registry server
logout	Log out from a Docker registry server
logs	Fetch the logs of a container
port	Lookup the public-facing port that is NAT-ed to PRIVATE_PORT
pause	Pause all processes within a container
ps	List containers
pull	Pull an image or a repository from a Docker registry server
push	Push an image or a repository to a Docker registry server
rename	Rename an existing container
restart	Restart a running container
rm	Remove one or more containers
rmi	Remove one or more images
run	Run a command in a new container
save	Save an image to a tar archive
search	Search for an image on the Docker Hub
start	Start a stopped container
stats	Display a stream of a containers' resource usage statistics
stop	Stop a running container
tag	Tag an image into a repository
top	Lookup the running processes of a container
unpause	Unpause a paused container
version	Show the Docker version information
wait	Block until a container stops, then print its exit code

Run 'docker COMMAND --help' for more information on a command.

这么多命令，让初学者一脸茫然，到底从哪个开始？别急，不需要全部掌握，我们常用的也就那么几个。

1. 显示 Docker 信息

使用“docker version”命令可以显示版本信息，运行结果如下：

```
$ docker version
Client version: 1.6.2
Client API version: 1.18
Go version (client): go1.2.1
```

```
Git commit (client): 7c8fca2
OS/Arch (client): linux/amd64
Server version: 1.6.2
Server API version: 1.18
Go version (server): go1.2.1
Git commit (server): 7c8fca2
OS/Arch (server): linux/amd64
```

可见，该命令执行结果显示了当前安装的 Docker 客户端/服务器软件版本、API 版本、Go 语言版本等信息。

如果需要了解当前 Docker 的使用状态，使用“docker info”命令：

```
$ docker info
Containers: 12
Images: 11
Storage Driver: aufs
Root Dir: /var/lib/docker/aufs
Backing Filesystem: extfs
Dirs: 35
Dirperm1 Supported: false
Execution Driver: native-0.2
Kernel Version: 3.13.0-83-generic
Operating System: Ubuntu 14.04.4 LTS
CPUs: 28
Total Memory: 62.79 GiB
Name: XXX
ID: XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX
WARNING: No swap limit support
```

该命令显示了当前容器、镜像数目信息，以及存储空间占用情况，还有操作系统内核版本、发行版本、硬件资源等信息。这些是系统管理员关心的信息。

2. 下载 Docker 镜像

下面我们运行带 pull 选项的 docker 命令，拉取一个镜像，即从 Docker 注册服务器的软件仓库下载一个现成的 Docker 镜像。

使用的命令如下：

```
$ docker pull ubuntu
Pulling repository ubuntu
```

```

4ef6a5ece191: Download complete
41fab45213e8: Download complete
cc592d60c68d: Download complete
02e32c081c51: Download complete
19284924629c: Download complete
Status: Downloaded newer image for ubuntu:latest

```

拉取镜像成功后，使用“docker images”命令查看系统中已有的镜像：

```

$ docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	latest	4ef6a5ece191	5 days ago	120.1 MB
hello-world	latest	f1d956dc5945	4 days ago	967 B

可以看到有两个镜像：ubuntu 和 hello-world，前者是刚刚运行“docker pull ubuntu”命令下载得到的；而 hello-world 是前面测试 Docker 安装是否成功时运行“docker run hello-world”命令下载得到的。

3. 从镜像创建 Docker 容器

利用下载好的镜像，可以使用“docker run”命令创建一个活动的容器：

```

$ docker run -i -t ubuntu/bin/bash
root@e6fb03f84b6e:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys
tmp usr var
root@e6fb03f84b6e:/# who
root@e6fb03f84b6e:/# uname -a
Linux e6fb03f84b6e 3.13.0-83-generic #127-Ubuntu SMP Fri Mar 11 00:25:37 UTC 2016 x86_64
x86_64 x86_64 GNU/Linux

```

“docker run”的参数解释如下：

- -i, 交互模式，让输入输出都在标准控制台进行；与此相反的是-d 选项，容器启动后进入后台运行，非交互模式。
- -t, 为新创建的容器分配一个伪终端（tty）。
- ubuntu, 这里给出用于创建容器的镜像名称，也可以用镜像 ID 代替。
- /bin/bash, 在新建容器中运行的命令，可为任意 Linux 命令。

TIPS: 举几个在 Docker 容器中运行常用 Linux 命令的例子:

```
$ docker run -i -t ubuntu hostname
4902514abb53
$ docker run -i -t ubuntu ls
bin dev home lib64 mnt proc run      srv tmp var
boot etc lib      media  opt root sbin sys usr
$ docker run -i -t ubuntu date
Sun May 1 12:37:52 UTC 2016
```

在容器中, 有时希望回到宿主机环境, 即临时断开容器和当前终端的连接, 可以使用组合键 “Ctrl + P” 和 “Ctrl + Q”, 返回宿主机终端会话:

```
root@610f016981be:/# 【先按下 “ Ctrl + P ”, 再按下 “ Ctrl + Q ”】
$ 【这里是宿主机终端会话】
```

在宿主主机上干了点事情之后, 想回到之前的容器 (不是新建一个容器), 怎么操作呢? 我们需要运行 “docker ps” 命令查看当前活动 (运行) 的容器:

```
$ docker ps
CONTAINER ID  IMAGE                COMMAND                  CREATED        STATUS        PORTS NAMES
610f016981be  ubuntu:latest        "/bin/bash"             26 minutes ago Up 26 minutes prickly_wilson
```

该命令打印当前活动容器的信息, 包括容器 ID、从哪个镜像创建的、容器中执行的命令、创建时间、状态、端口占用情况、容器名。通过该命令可以获得非常关键的信息, 我们后面会使用这些信息创建新的镜像。

可以看到活动容器的 ID 为 610*, 通过 “docker attach” 命令可以再次建立该容器和当前终端的连接:

```
$ docker attach 610
【看上去不明显, 实际已进入容器】ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys
tmp usr var
root@610f016981be:/#
```

命令行中的 “610” 就是容器 ID 的前 3 个字符。Docker 的一个方便之处在于不需要输入全部 ID, 而只输入前 3 个字符就能自动识别。后面使用的镜像 ID 也是如此。

4. 从容器创建 Docker 镜像

创建 Docker 镜像的方法主要有两种: 一种是从运行的容器创建镜像; 另一种是编写

DockerFile 文件创建镜像。这里我们介绍第一种方法，而第二种方法留为作业。

使用“docker commit”命令实现将运行的容器打为镜像，这个功能非常重要。

```
root@610f016981be:/# touch README.md
root@610f016981be:/# echo "This is a test modification" >> README.md
root@610f016981be:/# 【先按下“Ctrl + P”，再按下“Ctrl + Q”】
$ 【这里是宿主机终端会话】
$ docker commit -m "Test a change" 610 ubuntu:test_change
679acd7c336675d1cf22c27ab9a96cd782f6af472141e55e76fe5cc04d343638
```

上面的“docker commit”命令类似于 git 中的 commit 命令，提交更改到本地库。“610”为用来创建镜像的活动容器 ID，“ubuntu:test_change”为新镜像的库名：标签，我们查看当前系统中已有的镜像：

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
ubuntu	test_change	679acd7c3366	5 seconds ago	120.1 MB
hello-world	latest	f1d956dc5945	4 days ago	967 B
ubuntu	latest	4ef6a5ece191	5 days ago	120.1 MB

可以发现增加了一个新的镜像，既可以用 ubuntu:test_change 来标记，也可以用其镜像 ID (679) 来标记。

利用该镜像创建新的容器时，容器内包含了“docker commit”前所有更改的内容，我们测试一下：

```
$ docker run -t -i 679 /bin/bash
root@2231483a0ef0:/# cat README.md
This is a test modification
```

可见容器内包含了新建的文件“README.md”，实现了预期的更改。

5. 上传 Docker 镜像

如果希望将本地更改过的镜像分享给其他人，则需要通过“docker push”将本地镜像推送到 Docker Hub，该动作与“docker pull”刚好相反。

为了获得推送权限，你需要注册 Docker Hub 账号^[3]。注册成功后，运行“docker login”登录：

```
$ docker login
Username: zhaoyongke
```

```

Password:*****
Email: zhaoyongke@yeah.net
WARNING: login credentials saved in /home/zyk/.dockercfg.
Login Succeeded

```

推送本地镜像到 Docker Hub:

```

$ docker tag 679 zhaoyongke/mastercaffein21days:test_ubuntu_change
$ docker push zhaoyongke/mastercaffein21days:test_ubuntu_change
The push refers to a repository [zhaoyongke/mastercaffein21days] (len: 1)
679acd7c3366: Image already exists
4ef6a5ece191: Image successfully pushed
19284924629c: Image successfully pushed
02e32c081c51: Image successfully pushed
cc592d60c68d: Image successfully pushed
.....【等待时间较长，不建议读者推送到官方 Docker Hub，而是参考下节内容】

```

17.2.4 Docker 使用进阶

1. 使用阿里云 Docker Hub 管理镜像

考虑到国内读者访问 Docker 官网速度较慢，推荐使用阿里云 Docker Hub^{[4][5]}，如图 17-4 所示。



图17-4 阿里云Docker Hub

单击右上角的“管理中心”按钮，弹出登录界面，用你的阿里云账号登录，成功后显示如图 17-5 所示。



图17-5 阿里云Docker Hub管理中心

选择左侧的“镜像管理”标签，新用户会提示设置自己的镜像仓库“命名空间”和密码。命名空间是私人镜像的地址前缀。单击“立即设置”，本书这里设置为 `master_caffe_in_21days`，在设置命名空间的同时需要设置密码，请牢记该密码。

完成命名空间和密码设置之后，可以创建第一个镜像仓库（repository）。我们创建一个名称为 `caffe` 的镜像仓库，设置如图 17-6 所示。



图17-6 创建第一个Docker镜像仓库

“仓库类型”选择“公开”时，所有阿里云用户都能访问（只读）你的镜像仓库；而选择“私有”时，只有自己可见，读者应根据实际情况选择，避免将带有登录账号、包含商业代码、未清理使用记录的镜像推送到公开镜像仓库。

创建成功后回到管理中心，看到显示如图 17-7 所示。



图17-7 Docker镜像仓库创建成功

从图 17-7 看到我们刚刚创建的仓库地址为 `registry.aliyuncs.com/master_caffe_in_21days/caffe`，这是个公网地址，也称为外网地址。在阿里云 ECS 或 HPC 物理机上可以使用内网地址获得极高的内网带宽，内网地址为 `registry-internal.aliyuncs.com/master_caffe_in_21days/caffe`。

使用阿里云账号和之前设定的 Docker 仓库密码，登录阿里云 Docker 仓库。

公网用户：

```
$ docker login registry.aliyuncs.com
Username: zhaoyongke@yeah.net
Password: *****
Email: zhaoyongke@yeah.net
WARNING: login credentials saved in /home/XXXX/.dockercfg.
Login Succeeded

$ docker tag 679 registry.aliyuncs.com/master_caffe_in_21days/caffe:base
$ docker push registry.aliyuncs.com/master_caffe_in_21days/caffe:base
The push refers to a repository [registry.aliyuncs.com/master_caffe_in_21days/caffe] (len: 1)
679acd7c3366: Image already exists
4ef6a5ece191: Image successfully pushed
19284924629c: Image successfully pushed
02e32c081c51: Image successfully pushed
cc592d60c68d: Image successfully pushed
41fab45213e8: Image successfully pushed
Digest: sha256:990c55193fae00b9dff22fd4e9626eabf4ba132aac9ce596b86726d33e769c95
```

在阿里云控制台查看镜像版本，如图 17-8 所示。

操作指南		镜像版本	
Image ID	版本	最后构建时间	sha256
a31417f14025	base	2016-05-02 19:43:53	990c55193fae00b9dff22fd4e9626eabf4ba132aac9ce596b86726d33e769c95

图17-8 Docker镜像推送成功后的控制台信息

TIPS: 内网用户可以将上述 Docker Hub 地址改为内网地址以获得更高的带宽：

```
# docker login registry-internal.aliyuncs.com
# docker tag imageidregistry-internal.aliyuncs.com/YOUR_NAMESPACE/YOUR_REPO
# docker push registry-internal.aliyuncs.com/YOUR_NAMESPACE/YOUR_REPO
```

此时读者已经在阿里云 Docker Hub 上发布了第一个 Docker 镜像^[6]，相当于已将开发测试环境打包到镜像中。在另一台机器上部署该镜像非常简单，首先将镜像拉取到本地：

```
$ docker pull registry.aliyuncs.com/master_caffe_in_21days/caffe:base
Trying to pull repository registry.aliyuncs.com/master_caffe_in_21days/caffe ... base:
Pulling from master_caffe_in_21days/caffe
679acd7c3366: Already exists
41fab45213e8: Already exists
cc592d60c68d: Already exists
02e32c081c51: Already exists
19284924629c: Already exists
4ef6a5ece191: Already exists
Digest: sha256:990c55193fae00b9dff22fd4e9626eabf4ba132aac9ce596b86726d33e769c95
Status: Downloaded newer image for registry.aliyuncs.com/master_caffe_in_21days/caffe:base
```

然后利用该镜像创建容器：

```
$ docker run -ti registry.aliyuncs.com/master_caffe_in_21days/caffe:base /bin/bash
root@b38f60072664:/# ls
README.md bin boot dev etc home lib lib64 media mnt opt proc root run sbin
srv sys tmp usr var
root@b38f60072664:/# cat README.md
This is a test modification
root@b38f60072664:/#
```

这样就实现了利用 Docker 从开发机器向生产机器的迁移部署。

2. 在 Docker 容器中访问 GPU

笔者已经在阿里云容器 Hub 上发布了基于 Ubuntu 14.04 系统并部署好 Caffe + CUDA 7.5 + GPU 驱动 352.39 环境的镜像^[6]，读者可在支持 Docker 的系统中执行“docker pull”命令获得该镜像：

```
$ docker pull registry.aliyuncs.com/est123/digits2:caffe
Trying to pull repository registry.aliyuncs.com/est123/digits2 ... caffe: Pulling from
est123/digits2
d3a1f33e8a5a: Pull complete
c22013c84729: Pull complete
d74508fb6632: Pull complete
91e54dfb1179: Pull complete
37cb59f61a4d: Pull complete
8991059993d2: Pull complete
12c38bd7b0da: Pull complete
cfa6969bb539: Pull complete
a50ed1359171: Pull complete
5f71a95c0a9f: Pull complete
Digest: sha256:9960d20784617db3e6cb6b423c15f83f43c4cf3ef42c7a9a6c53ce6a8a4d5395
Status: Downloaded newer image for registry.aliyuncs.com/est123/digits2:caffe
```

查看镜像：

```
$ docker images
```

REPOSITORY	TAG	CREATED	IMAGE ID	VIRTUAL SIZE
registry.aliyuncs.com/est123/digits2	caffe	5f71a95c0a9f	9 weeks ago	4.813 GB

这时使用前面的“docker run”命令创建容器，在运行 Caffe 时 CUDA 会报错。需要利用如下命令创建容器：

```
$ docker run -ti \
--device /dev/nvidia0:/dev/nvidia0 \
--device /dev/nvidiactl:/dev/nvidiactl \
--device /dev/nvidia-uvm:/dev/nvidia-uvm \
-v /disk1:/disk1 \
5f7 /bin/bash
```

其中，选项--device 实现了容器直通宿主机的相应设备，这样在容器中就能正常运行 CUDA 程序了。读者可以使用 15.2.4 节中的方法验证容器中 GPU 驱动与 CUDA 是否安装成功。

如果 GPU 驱动和 CUDA 测试没有问题,则可以进入/root/caffe/目录,运行如下命令进行测试:

```
# ./build/tools/caffe.bin time -model models/bvlc_reference_caffenet/deploy.prototxt  
-gpu 0
```

运行输出请参考第 15 天内容。

至此,读者掌握了 Docker 工具,可以获得更大的自由选择其他支持 Docker 的深度学习开发工具(如 TensorFlow)。使用 Docker 可以让你养成保持干净环境的好习惯,避免无意识地破坏已有的开发工作,提高产品迭代、发布效率,适应越来越快的互联网的工作节奏,成为优秀的生产工程师(PE)。

17.3 练习题

1. 学习 Docker File 创建镜像。
2. 容器技术和虚拟机技术有哪些相同点与不同点?
3. Docker 容器内能否再运行 Docker 容器?
4. 分别在 Docker 容器和宿主机上运行相同的计算程序,对比并评估 Docker 容器引起的性能损失情况。
5. 获取其他深度学习框架的 Docker 镜像。

17.4 参考资料

[1] How to run the Caffe deep learning vision library on Nvidia's Jetson mobile GPU board, Pete Warden's blog, <https://petewarden.com/2014/10/25/how-to-run-the-caffe-deep-learning-vision-library-on-nvidias-jetson-mobile-gpu-board/>

[2] Docker 官网: <https://www.docker.com/>

[3] Docker Hub: <https://hub.docker.com/>

[4] 阿里云 Docker Hub: <https://dev.aliyun.com/search.html>

[5] 阿里云 Docker 使用文档: <https://hpc.aliyun.com/doc/docker> 镜像服务

[6] 阿里云深度学习工具集: <http://dev.aliyun.com/detail.html?repoId=2>

第 18 天

关于 ILSVRC 不得不说的一些事儿

前面多次提到 ILSVRC (ImageNet Large Scale Visual Recognition Challenge, “图网”大规模视觉识别竞赛), 世界上最顶尖的深度学习研究人员、企业都在这场竞赛中角逐, 不断刷新图像识别纪录。今天我们将目光聚焦于这项赛事, 读者可以通过今天的内容了解比赛规则并吸取历年各项任务冠军的经验, 让自己近距离接触国际领先技术, 指导今后的学习和工作。

ILSVRC 自从 2010 年开始, 每年举行一次。具有数百个物体类别、百万量级图片, 成为计算机视觉领域图像分类、检测算法性能评估参考标准。

18.1 ImageNet 数据集

ImageNet 数据集是按照 WordNet 架构组织的大规模带标签图像数据集, 其发起者为斯坦福大学教授李飞飞^[1]。图 18-1 显示了 ImageNet 中一些样例图片。



图18-1 ImageNet样例图片

在 WordNet 中有意义的概念 (可能是多个单词或短语) 称为一个同义词集 (synonym set, 简称 synset)。在 WordNet 中有超过 100 000 个同义词集, 绝大部分是名词。ImageNet 选取了

WordNet 中 21 841 个同义词集并为每个同义词集提供了平均 650 张全分辨率图片，总共 14 197 122 张标注图片，每张图片都经过了严格人工筛选与标记，这是一项极具挑战性的工作。与其他图像分类数据集相比，ImageNet 具有规模大、多元化等优势。

读者需要注意，ImageNet 并没有这些图像的所有权，只是提供了这些图像的索引和 URL，类似于图像搜索引擎，所以在商业应用中使用这些图像有一定的风险。

18.2 ILSVRC 比赛项目

ILSVRC 延续了 PASCAL VOC 比赛项目^[2]，后者从 2005 年开始，以一年一度的比赛形式形成了评估图像识别算法的标准化方法。

ILSVRC 主要由“公开数据集 + 年度比赛和专题研讨会”两部分构成，公开数据集用于开发 and 对比目标分类识别算法，比赛和专题研讨会提供每年跟踪最新进展和讨论从中学到的教训。

公开数据集包括人工标注的训练集，评测集也做了人工标注但标签不公开。参赛者使用训练集图片和标签训练其算法，然后自动标注评测集图片，将标注后的结果提交到比赛阶段开放的评价服务器，评价结果将在比赛结束阶段公布。作者将被邀请到国际计算机视觉会议 (ICCV)/欧洲计算机视觉会议 (ECCV) 中的专题研讨会分享其技术。

ILSVRC 使用的公开数据集是 ImageNet 的子集，就以 2012 年 ILSVRC 分类任务数据集 (Caffe 训练图像分类模型时通常用该数据集作为基准) 为例，其中训练集为 1 281 167 张图片 + 标签，验证集为 50 000 张图片 + 标签，用于最终打分的评测集为 100 000 张图片 (无标签)，这些图片属于 1 000 个不同类别。

有条件的话，也可以在完整的 ImageNet 数据集上训练，只是需要更大的硬盘空间，以及更长的训练时间。

每年的比赛流程大体为：

- 发布训练数据集 (图像+标注)。
- 发布测试数据集 (仅图像，标注隐藏)。
- 参赛者在训练数据集上训练模型。
- 以文本文件形式提交测试集上的预测结果。
- 组委会评估各个参赛者提交的结果，公布排名，开研讨会。

下面我们逐一介绍 ILSVRC 的具体比赛项目。

18.2.1 图像分类 (CLS)

图像分类 (CLS) 任务在 2010—2014 年的 ILSVRC 比赛中为独立任务，而从 2015 年比赛开始该任务与目标定位 (LOC) 任务合并。

在该任务中，算法输入为一张图片，输出为该图片中可能存在的物体类别信息。使用 Top-5 评估方式时，输出 5 个预测类别信息，只要真实类别在其中就算正确分类，否则记错误分类。对全部 100 000 张评测集图片统计错误分类样本数目，计算错误率。图 18-2 显示了 2010—2015 年图像分类任务错误率情况。

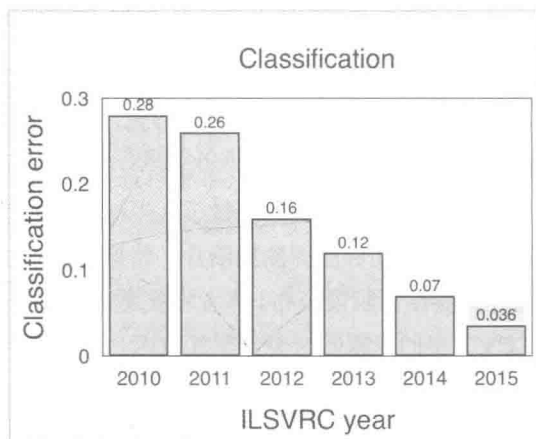


图18-2 2010—2015年分类任务错误率情况

18.2.2 目标定位 (LOC)

目标定位 (LOC) 任务从 2011 年开始登上舞台，到 2015 年与 CLS 任务合并为 CLS-LOC。

该任务的目的是评估算法能学习目标物体特征而不是背景。

单目标定位任务的数据与 CLS 任务包含相同的照片，手动标注图片中是否存在 1000 个物体类别之一的实例。每张图片包括一个真实标签。该类别的每个实例都标注了边界框。

对于每张图片，算法应产生一系列图像中存在的物体类别以及对齐的边界框用于指示物体位置和尺度，每个边界框中只能包含一个物体实例。根据物体类别标签和真实标签的最佳匹配程度评估标签质量，另外需要保证预测的实例位置也是正确的。图 18-3 显示了 2011—2015 年目标定位任务错误率情况。

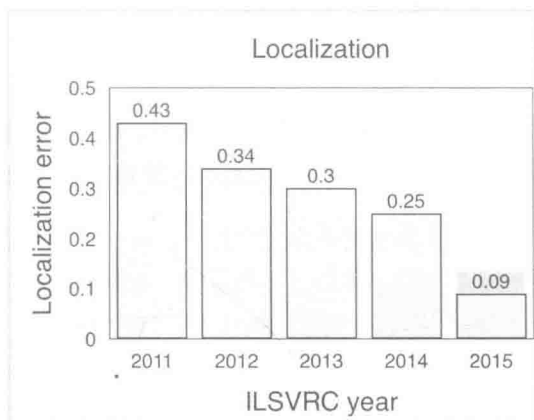


图18-3 2011—2015年目标定位任务错误率情况

18.2.3 目标检测 (DET)

目标检测 (DET) 任务在单物体定位 (LOC) 任务的基础上更进一步, 用于在图片中定位多个类别物体。

该任务是 PASCAL VOC 的一部分, 包含 200 个物体类别和数万张照片。对每张图片, 算法产生边界框, 指示所有目标类别的所有实例的位置和尺度。标签质量使用两种方式评估: 检出目标物体实例数和检出精度, 或者算法产生的虚假检测数。图 18-4 显示了 2013—2015 年目标检测任务平均准确率。

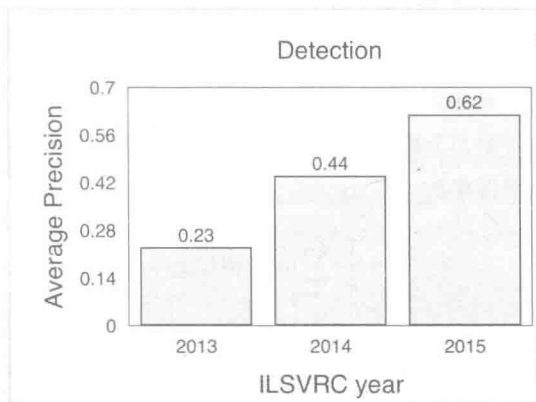


图18-4 2013—2015年目标检测任务平均准确率

18.2.4 视频目标检测 (VID)

ImageNet 视频目标检测 (VID) 任务是 2015 年引入的尝鲜赛，类似于从静态图像中进行目标检测 (DET) 任务。该任务的数据集包括 30 个类别，是 DET 任务 200 个类别的子集。

每个视频片段的所有帧、所有类别物体都完全标注。这些类别都是精心选择的，考虑到不同因素，如运动类型、视频背景干扰、平均目标数目等。对每个视频片段，算法需要产生一组标注信息 (f_i, c_i, s_i, b_i) ，表示第 f_i 帧、第 c_i 个类别、置信度 s_i 和边界框 b_i 。评估准则和 DET 任务相同，该集合每帧都包含 30 个目标类别中的某个实例。评估准则和目标检测任务相同，未标记的目标将被惩罚，重复检测也会被惩罚。在大多数目标类别中获得最高准确率的将会胜出。

使用常规的平均准确率 (mean Average Precision, mAP) 在所有类别上平均作为评估准则。

18.2.5 场景分类

场景分类任务也是 2015 年引入的尝鲜赛，由 MIT Places 研究组组织。该任务的目的是识别照片中描述的场景类别，数据来源于 Places2 数据集（共包括 1000 多万张图像，属于 401 个场景类别）。特别地，比赛数据分为 810 万张训练图像、2 万张验证图像和 38.1 万张测试图像，均属于 401 个场景类别。注意不同类别的图像分布不均匀（从 4000 张到 30000 张不等），正如这些场景在现实中出现的频率一样。

对于每张图片，算法应产生 5 个场景类别（按照置信率降序排列）的列表，标签质量将使用图片最佳匹配真实标签评估。允许一个算法对一张图片识别多个场景类别，因为很多环境有多个标签（一个酒吧也是一个餐馆），人们也常常用不同的词语描述同一个地方（如森林、树丛）。

对于每张图片，算法产生 5 个标签 $l_j, j=1,2,3,4,5$ ，该图片的真实标签是 $g_k, k=1, \dots, n$ ，其中 n 为场景类别数。该算法的错误率为：

$$e = \frac{1}{n} \cdot \sum_k \min_j d(l_j, g_k)$$

其中：

$$d(x, y) = \begin{cases} 0, & x = y \\ 1, & \text{others} \end{cases}$$

总偏差是针对整个测试集图片计算平均偏差。目前比赛版本 $n=1$ ，即每张图片只有一个真

实标签。

图 18-5 显示了场景分类错误率情况。

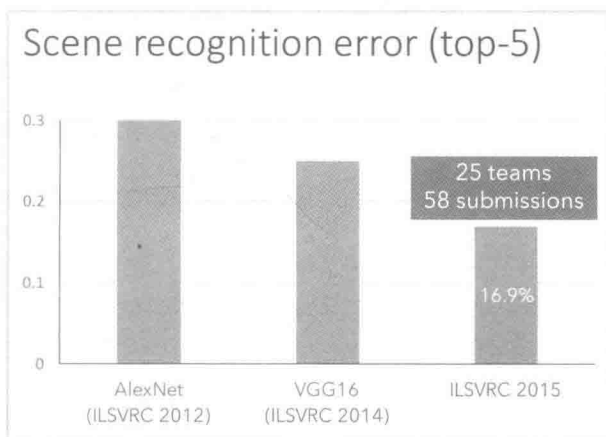


图18-5 场景分类错误率情况

18.3 Caffe ILSVRC 实践

假设已经下载了 ImageNet 训练数据和验证数据^[3]，它们存放在硬盘上，就像下面这样：

```
/path/to/imagenet/train/n01440764/n01440764_10026.JPEG
/path/to/imagenet/val/ILSVRC2012_val_00000001.JPEG
```

在 Caffe 源码框架的 data/ilsvrc12/下运行脚本来获取用于训练的额外数据：

```
$ ./get_ilsvrc_aux.sh
Downloading...
--2015-10-17 04:03:19-- http://dl.caffe.berkeleyvision.org/caffe_ilsvrc12.tar.gz
Resolving dl.caffe.berkeleyvision.org (dl.caffe.berkeleyvision.org)... 169.229.222.251
Connecting to dl.caffe.berkeleyvision.org (dl.caffe.berkeleyvision.org)|
169.229.222.251|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 17858008 (17M) [application/octet-stream]
Saving to: 'caffe_ilsvrc12.tar.gz'
```

```
100%[=====>] 17,858,008 75.8KB/s in 3m 0s
```

```
2015-10-17 04:06:20 (96.6 KB/s) - 'caffe_ilsvrc12.tar.gz' saved [17858008/17858008]
```

```

Unzipping...
Done.
$ tree
.
|-- det_synset_words.txt
|-- get_ilsvrc_aux.sh
|-- imagenet.bet.pickle
|-- imagenet_mean.binaryproto
|-- synsets.txt
|-- synset_words.txt
|-- test.txt
|-- train.txt
|-- val.txt
0 directories, 9 files

```

其中，训练和验证输入分别在 `train.txt` 和 `val.txt` 中描述，它们以文本方式列出了所有文件及其对应的标签。`synset_words.txt` 中记录了 `synset/name` 映射关系。在 `examples/imagenet/create_imagenet.sh` 中，设置正确的原始图像数据路径，设置 `RESIZE=true` 将所有图像缩放到 `256×256`。设置正确后，在 Caffe 源码根目录下执行 `./examples/imagenet/create_imagenet.sh`，将原始图像转换为 LMDB。

```

./examples/imagenet/create_imagenet.sh
Creating train lmdb...
I0301 13:37:40.202742 22009 convert_imageset.cpp:83] Shuffling data
I0301 13:37:41.517595 22009 convert_imageset.cpp:86] A total of 1281167 images.
I0301 13:37:41.518643 22009 db_lmdb.cpp:38] Opened lmdb examples/imagenet/
ilsvrc12_train_lmdb
I0301 13:38:15.292979 22009 convert_imageset.cpp:144] Processed 1000 files.
^^I0301 13:38:47.676054 22009 convert_imageset.cpp:144] Processed 2000 files.
I0301 13:39:18.747720 22009 convert_imageset.cpp:144] Processed 3000 files.
// .....预计需要 10 个小时左右才能完成

```

根据硬件情况不同，ILSVRC 2012 的训练、验证数据转换过程大约需要几个小时到十几个小时不等。

接下来需要计算图像均值，CNN 模型需要从每张输入图片中减去图像均值，所以需要预先计算图像均值。`tools/compute_image_mean.cpp` 实现了该功能。这也是一个很好的用来熟悉怎样

操作多个模块（ProtoBuffer、LEVELDB、GLOG 等）的例子。计算均值可直接在 Caffe 根目录下运行脚本：

```
./examples/imagenet/make_imagenet_mean.sh
```

生成的图像均值位于 data/ilsvrc12/imagenet_mean.binaryproto 中。

数据已经就绪，模型选择参考实现 models/bvlc_reference_caffenet/train_val.prototxt。

train 和 test 模型只有输入、输出层是不同的：

- 输入层——训练网络数据输入层从 ilsrvrc12_train_leveldb 中获取数据，并对输入图像采取随机镜像反转操作；而测试网络从 ilsrvrc12_val_leveldb 中获取数据，并不会对输入图像做随机镜像操作。
- 输出层——都使用 softmax_loss 层，训练阶段用于计算 loss 函数并初始化反向传播过程，而验证阶段直接打印输出 loss；测试网络还有额外的 accuracy 层，用于报告当前网络在验证集上的预测准确率。在训练阶段，测试网络隔一段时间发起并做一次测试，输出 Test score #0: xxx 和 Test score #1: xxx。在这里 score 0 是准确率（未训练网络从 $1/1000=0.001$ 开始），score 1 是 loss（未训练网络从 7 开始）。

求解器参数在 models/bvlc_reference_caffenet/solver.prototxt 中。

训练 ImageNet：

```
./build/tools/caffe train \
--solver=models/bvlc_reference_caffenet/solver.prototxt
```

在 K40 机器上，每 20 个批量数据（批量大小为 256）过网络时间为 26.5s（K20 需要 36s），所以训练阶段每张图片过网络时间大约为 5.2ms，这是完整的前向+后向所需时间。对于只有前向的情况，每张图片过网络时间大约为 2ms。

计时：

```
./build/tools/caffe time \
--model=models/bvlc_reference_caffenet/train_val.prototxt
```

从上一个快照处继续训练：

```
./build/tools/caffe train \
--solver=models/bvlc_reference_caffenet/solver.prototxt \
--snapshot=models/bvlc_reference_caffenet/caffenet_train_10000.solverstate
```

在 `caffenet_train_10000.solverstate` 中存储了所有必需的信息来恢复训练（包括学习速率、遗忘因子等训练参数和历史权值更新值）。

掌握今天的学习内容，读者若对该类比赛感兴趣，则可以试着个人或组队报名，通过实战提升自身水平，与世界上更多的深度学习爱好者同台竞技，学习到的内容会更多。全身心投入比赛中，有一个既定目标，能激发你更多的创造性想法。

18.4 练习题

1. 将自己手机中的照片做成训练/测试数据集，需要哪些工作？
2. Google、Apple、Microsoft、Amazon、BAT 等互联网公司如何获取业务相关的大规模图像数据？
3. 在 ImageNet 上训练收敛的模型，能否直接用于真实世界的图像分类或目标识别？

18.5 参考资料

[1] ImageNet Large Scale Visual Recognition Challenge, arXiv:1409.0575v3

[2] The PASCAL Visual Object Classes (VOC) Challenge. IJCV 2010

[3] 历年 ILSVRC 数据集下载链接：

<http://image-net.org/challenges/LSVRC/2015/download-images-3j16.php>

<http://image-net.org/challenges/LSVRC/2014/download-images-5jj5.php>

<http://www.image-net.org/challenges/LSVRC/2013/download-images-rpa>

<http://www.image-net.org/challenges/LSVRC/2012/nonpub-downloads>

<http://www.image-net.org/challenges/LSVRC/2011/registered-downloads>

<http://www.image-net.org/challenges/LSVRC/2010/download-all-nonpub>

第 19 天

放之四海而皆准

俗话说——学而不思成树袋，思而不学变民科。听了很多大道理，仍然过不好这一生。今天从实际业务：图像、文本、视频、艺术作品出发，重新审视深度学习是否如同传说中那么强大。

19.1 图像分类

本节我们先回顾一下计算机视觉领域中图像分类这个老问题（我们熟悉的 MNIST、CIFAR10、ImageNet 等例程全都属于该类问题）。图像分类问题看似简单，却是计算机视觉领域的一个核心问题，具有大量实际应用案例。很多典型的计算机视觉问题（如物体检测、图像分割）可以退化为图像分类问题^[1]。

19.1.1 问题描述

图像分类：给定一张输入图像，让机器判断它属于某固定类别集合中的具体哪一类。假设给 5 次猜测机会，只要 5 次中有一次猜对，就认为机器的判断是正确的，称为 Top-5 判据。从图像分类问题入手，符合“先说是不是，再问为什么”的优良传统。

图像分类问题虽然已经有大量研究，但是仍然有很多难点需要解决，主要有：

- 观测角度变化，相机相对一个观测物体改变角度时，会引起图像像素值剧烈变化（见图 19-1）。
- 光照条件变化，显然对于某一场景，有光和无光时图像像素值差异巨大（见图 19-2）。



图19-1 同一目标不同观测角度的图像

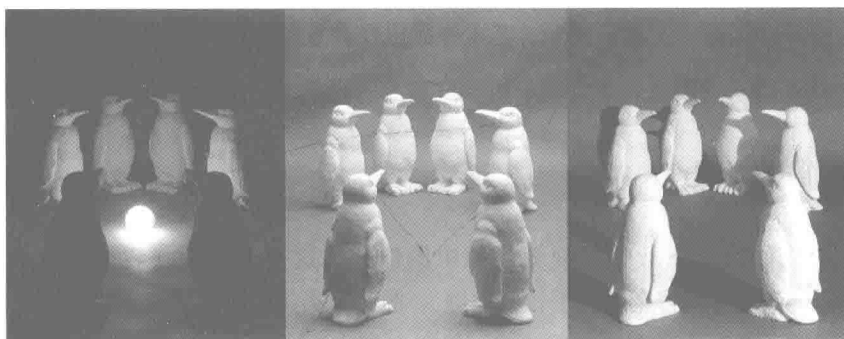


图19-2 同一场景不同光照下的图像

- 物体自身形变，很多被观测物体不是刚体，可能会发生极端变形（见图 19-3）。



图19-3 同一类别物体不同形态的图像

- 物体部分遮挡，被观测物体可能被遮盖一部分，有时只有很小部分可见（见图 19-4）。
- 背景杂波影响，被观测物体可能与周围环境融为一体，难以分辨（见图 19-5）。



图19-4 部分遮挡的目标图像

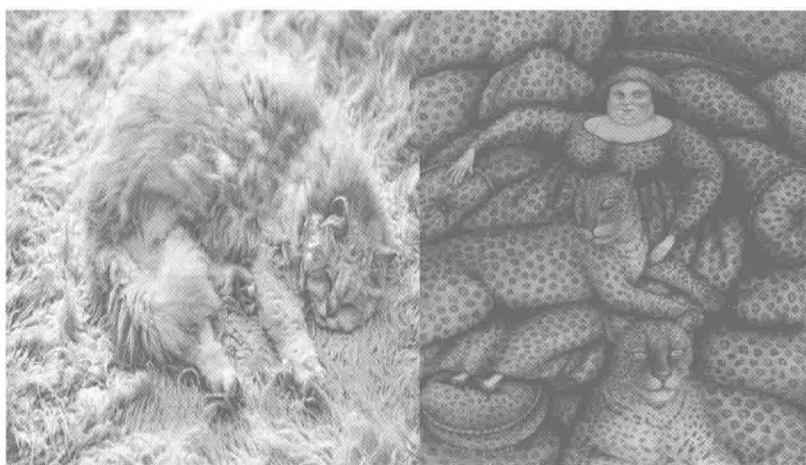


图19-5 有背景杂波影响的图像

○ 类内差异，被观测物体的同一类物体差异可能很大，“龙生九子，子子不同”（见图 19-6）。

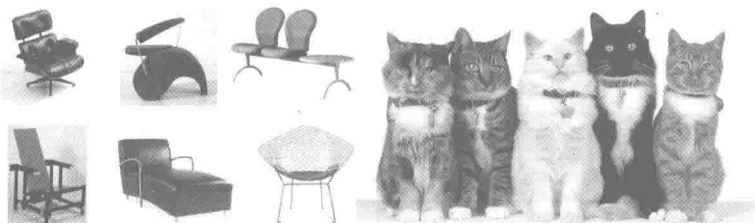


图19-6 类内差距明显的图像

以上几种情况都会导致被观测物体的计算机表示（二维或三维数值数组）发生剧烈变化。一个良好的图像分类模型应当对上述情况（以及不同情况的组合）不敏感，这是一个极具挑战性的任务。在深度学习出现之前，没有很好的解决方法。使用深度学习尤其是深度卷积神经网络，用大量图像数据进行训练后可以处理十分复杂的分类问题。我们来看一个具体应用案例。

19.1.2 应用案例——商品分类^[2]

女装作为淘宝和天猫平台的一大类目有其自身特色，买家和卖家更多的依赖图像去描述和选择商品，因此对女装图像分析有着实际的应用价值。其中类目预测是一个重要的功能，如图 19-7 所示，其过程是用户上传一幅服饰类图像，系统自动识别出服饰主体，并判别其所属的类目。

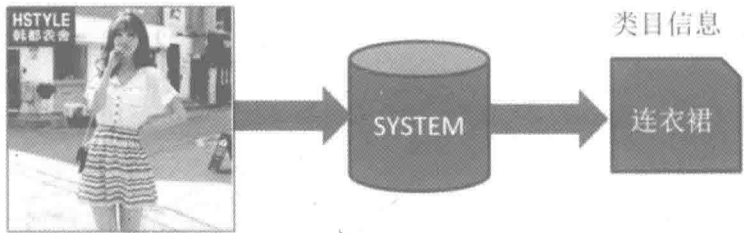


图19-7 女装类目识别

在我们的试验中，使用 110 万张淘宝女装图像作为训练数据，以及 5 万个验证集数据和 42 万个测试集数据。淘宝女装线上数据覆盖 27 个二级类目，部分类目重叠较多，因此对其进行合并，最终分成 17 个类目，包括连衣裙、衬衫、裤子、裙子、大码女装等，如表 19-1 所示。

表 19-1 用于试验的淘宝女装

编 号	类 目
0	裤子、牛仔褲
1	连衣裙
2	毛衣、毛针织衫
3	卫衣、情侣装
4	职业套装/学生校服/工作制服
5	大码女装、中老年女装
6	风衣
7	半身裙

续表

编 号	类 目
8	婚纱/礼服/旗袍
9	唐装/民族服装/舞台服装
10	棉衣、羽绒服
11	皮衣、皮草
12	外套、短外套、毛呢外套
13	T 恤
14	蕾丝衫、吊带
15	西装
16	衬衫

采用基于深度学习的图像分类方法训练女装分类的模型，主要过程包括：训练图像预处理，数据平衡化处理，使用一个 AlexNet 改进版网络结构在一块 Tesla K20 GPU 上经过大约 6 天的训练，类目预测模型在校验图像数据集上收敛到了 86% 的准确度。为了测试该模型在实际使用中的表现，使用一个与训练集不同的 42 万张淘宝女装图像作为测试集。测试结果如表 19-2 所示。

表 19-2 分类结果

评估方法	结 果
Top-1 准确率	87.4181%
Top-2 准确率	96.2241%
Top-3 准确率	98.4329%

在相同数据集上，我们现在获得的 Top-1 准确率较之前开发的基于 local feature + BOW 的传统算法，有较大的提高。经过测试，我们的算法对网上图和实拍图都有不错的识别效果。图 19-8 给出了两个预测结果示例。

本文摘选自阿里巴巴 ATA 博客。希望读者能借鉴其中思路，利用已有的 CNN 模型对自己收集的图像数据进行分类、预测，开发出更多好玩的应用。笔者能想到的有：

- 车辆型号识别，用手机对路上某辆车进行拍照，直接报告相应车辆品牌名、市场价。
- 树叶、花卉识别，拍照直接告诉你这是什么树，那是什么花，对于很多树痴、花痴非常有用。

	手机实拍图	网上图
		
预测结果	类目：连衣裙，79.68% 大码/中老年女装，9.38% T 恤，2.83%	类目：婚纱/礼服/旗袍，78.54% 唐装/民族服装/舞台服装，18.23% 连衣裙，1.31%

图19-8 预测结果

- 食物识别，对朋友圈那些晒美食的来一波致命打击，告诉他们这个有多少热量，需要跑多少圈才能消耗完毕。
- 茶叶、香烟、酒类识别，利于不擅长品茶、品烟、品酒的人装 b。
- 交通标志识别，这个数据可以从驾校参考书中获得，用于自动驾驶汽车。

19.2 图像中的字符识别

19.2.1 问题描述

光学字符识别（Optical Character Recognition, OCR）是指对文本资料的图像文件进行分析识别处理，获取文字及版面信息的过程。例如，用手机拍下一些文字信息（可能是上课老师黑板上的文字或图书馆一本书的内容）保存为图片，将图片转换为文本文件就需要使用 OCR 技术。OCR 的目的是把图像中的文字识别出来，以便让计算机进一步利用。

OCR 并不是一项新技术，最早用模板匹配来进行数字和英文字母识别的专利可追溯至 1929 年，而汉字识别则始于 20 世纪 60 年代。OCR 算法技术，可以分为两个发展阶段：基于传统方法和基于深度学习方法。深度学习应用于 OCR 大大提高了 OCR 的识别性能，但在 OCR 识别流程中，深度学习还不能完全代替传统方法，如中文的文本检测、切分等模块。深度学习虽然有识别率高、泛化能力强等优点，但在实际研发和应用中，其缺点也是明显的，一是解码速度慢；二是需要的训练样本多。

19.2.2 应用案例——身份证实名认证^[3]

OCR 技术最初用于邮政系统的邮编数字识别，以实现邮件自动分拣功能。随着光学成像设备（扫描仪、数码相机、手机等）的飞速发展，OCR 也获得了广泛的应用。例如，2010 年第六届全国人口普查就使用 OCR 技术对约 8 亿份表单进行了自动信息录入的操作，极大地降低了人力成本。近年来，OCR 更加受到工业界的关注，例如 Google 和百度利用深度学习技术，研发了自然场景文字识别技术，能够自动检测并识别出图片中的文字信息。

身份证实名认证是 OCR 领域中的一个具体应用，由于二代身份证是全国统一的，格式单一、字体固定、字符间距固定，所以，在算法上，是 OCR 中相对简单的一种，可以通过结合先验知识，把识别率做得很高。身份证 OCR 识别流程如图 19-9 所示。为了保证高识别率和高速度，我们的设计思路同时融合了传统算法和深度学习算法。

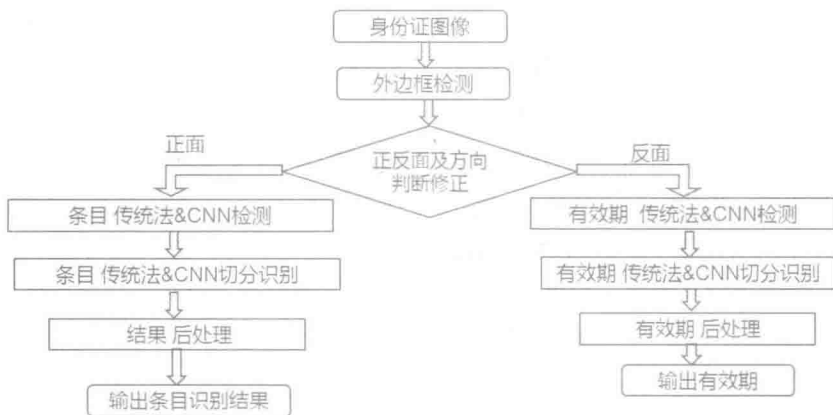


图19-9 身份证OCR识别流程

1. 外边框检测

由于身份证在图像中的占比和位置不定，为了使接下来的处理更加精准，首先需要定位身份证的外边框位置。这样，接下来的处理只集中于身份证图像本身，而排除背景图像干扰。

- 算法/模型：基于 CNN（卷积神经网络，一种深度学习网络）的外边框定位
- 训练图像样本：5 万张左右
- 定位准确度：99.90%
- 速度：平均 25ms/image

2. 正反面及方向判断和修正

虽然实人认证系统界面提示用户把身份证正反面要分开上传，但总有用户不按要求做，导致正反面放反的情况时有发生。同时，证件在图像中也可能是倒着的或旋转 90° 的，甚至是斜着放置的。所以，这里对这些情况进行自动判断（正反面、放置方向）。

- 算法/模型：基于 CNN 的正反面及方向判断，并根据结果进行正反面分类和方向修正
- 训练图像样本：10 万张左右
- 定位准确度：99.82% 以上
- 速度：平均 30ms/image

3. 条目检测

身份证识别的条目包括正面的姓名、身份证号、地址和背面的有效期。其中地址的检测还包括行的检测，因为地址可能包括一行或多行。条目检测过程实际上包括文本/非文本的判断和文本的定位两个过程。文本的检测是 OCR 领域中一个比较困难的问题。不过，由于身份证是一个垂直领域，有很多先验知识可以利用，所以，变得相对简单很多。在实际输入中，大多数图像都比较清晰可辨，用传统的方法就能检测得很好。传统方法最大的优点是速度快，并且需要用来训练的样本数据很少，但对于比较模糊或反光或不全的图像，传统方法往往就无能为力了。这时，需要融合深度学习方法，利用 CNN 强大的学习和泛化能力就可以很好地解决。

- 基于传统算法的条目检测：检测过程是图像归一化→二值化→连通域分析→字符块/非字符块判定→基于规则的条目定位→地址行切分
- 基于深度学习（CNN）的条目检测：对正反面条目分别训练检测模型，训练图像样本 20 万张左右
- 检测精度：99.89%
- 速度：背面平均 266ms/image；正面 135ms/image

4. 字符的切分和识别

字符的切分和识别是身份证 OCR 中最重要的算法模块之一。同样，对于清晰、对比度好的图像，用传统算法就可以切分/识别得很好；但对于模糊、反光、变形的图像，深度学习算法明显更胜一筹。并且，对于切分正确的单字符识别，基于深度学习（CNN）模型的方法好于基于模板字符内核的方法。当然，前者速度明显慢于后者。考虑到身份证的字符较少，每个字符速

度慢一些，影响不大，所以，OCR 字符识别采用了基于 CNN 的识别模型。

- 基于传统算法的字符切分：字符切分过程是条目区域二值化→连通域分析与过滤→切分结果
- 基于深度学习（CNN）算法的字符切分/识别：对汉字和数字分别训练切分和识别模型
 - 深度学习包括的字符集：共 5454 个字符。通过对 2000 万个身份证地址和姓名进行统计，5454 个字符可覆盖总字频数的 99.9938%
 - 训练单字符样本数：1 亿左右
 - 识别速度：汉字平均 3ms/字符，数字和英文平均 1ms/字符
 - 基于传统算法和深度学习（CNN）算法的字符切分/识别有机融合后，引擎具备了十分强大的识别和泛化能力。

5. 后处理

（1）姓名识别的后处理

主要处理把识别结果中出现的一些繁体汉字和非常见汉字映射为常见的姓名汉字，以及对一些非姓汉字的映射。

（2）有效期识别的后处理

利用有效期起始日期的识别结果，对截止日期识别结果中部分缺失数字进行补齐、部分错误数字进行纠正，以及对一些特殊位的数字进行映射。

（3）身份证号识别的后处理

18 位身份证号是有固定格式的，如：前 6 位是行政区划码，接着的 8 位是出生日期。身份证地址识别出来后，能解析出地址对应的 6 位行政区划码，这样就能对身份证号码的前 6 位进行校正了。出生日期的 8 位数字是有一定规律的，也能利用识别结果的候选和置信度进行一定程度的修正。

（4）地址识别的后处理

首先，建立中国的地址信息库，对置信度不高的识别结果进行替换修正；其次，实名认证业务应用需要知道身份证上地址对应的每一级行政区划，且需要是最新的行政区划，包括省、

市、县（区）三级，而身份证上的地址信息往往是有部分缺失的，这样，OCR 引擎需要从地址的识别结果中恢复所缺失的行政区划。对应的算法是依据国家统计局的新旧行政区划表来设计完成的。

6. 结论与展望

通过上述步骤对一些典型的身份证数据进行识别，结果如图 19-10 和图 19-11 所示。

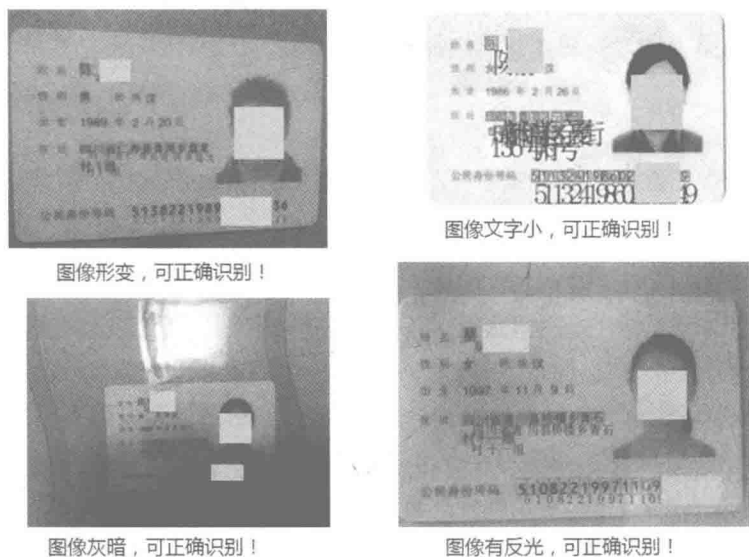


图19-10 可正确识别的正面图例

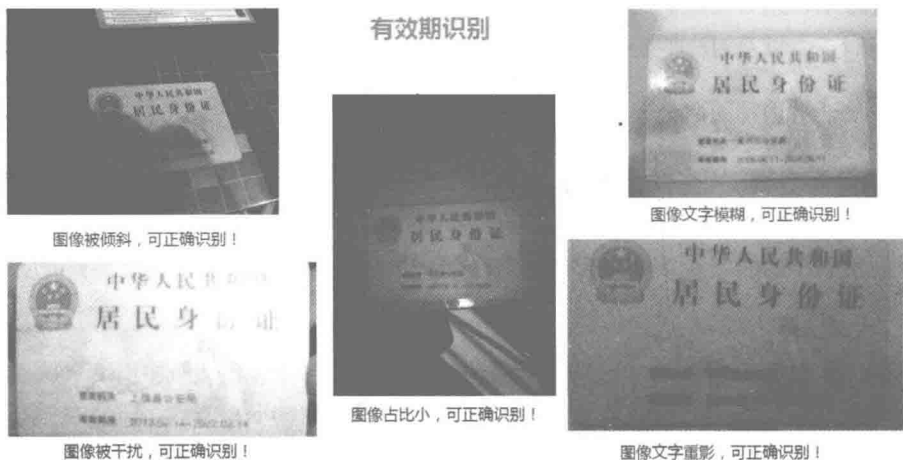


图19-11 可正确识别的反面图例

本节介绍的方法也可以应用到其他证件的 OCR 识别中，例如驾驶证、行驶证、营业执照、护照等。读者不妨利用已有的数据和模型进行尝试。

19.3 目标检测

19.3.1 问题描述

目标检测（Object Detection）是机器视觉中最常见的问题。当机器“睁”开双眼看世界时，需要判断其视野内存在哪些目标、分别是什么、在什么位置。机器接收的数据来源可能为静态图像（手机相机）或视频（监控摄像头），对于目标检测算法而言略有不同。

近两年来，目标检测算法的性能不断提升，除了引入强大的深度神经网络之外，最新的目标检测框架如 R-CNN^[4]及其后代 Fast R-CNN^[5]和 Faster R-CNN^[6]也扮演了相当重要的角色。它们对静态图像是非常有效的，但这些框架并不是针对视频目标检测特别设计的，没有完全利用视频的时域和上下文信息。参考资料[7]中提出了一种深度学习框架 T-CNN，从视频中获得 tubelets 中集成了时域和上下文信息，显著改善了视频中目标检测性能，赢得了 ILSVRC 2015 最近引入的 VID 任务（只用提供的数据）冠军。

19.3.2 最佳实践——运行 R-CNN 例程

运行 R-CNN 例程需要配置 MatCaffe 环境，具体步骤可参考 16.2.2 节。

R-CNN 依赖 Caffe v0.999，需要从 <https://github.com/BVLC/caffe/archive/v0.999.tar.gz> 下载并修改 Makefile.config、编译，具体步骤请参考第 4 天介绍的内容。将 Caffe v0.999 所在目录记为 \$CAFFE_ROOT。预先下载 ImageNet 图像均值文件：

```
$ cd $CAFFE_ROOT/data/ilsvrc12 && ./get_ilsvrc_aux.sh
```

切换到另一个工作目录，获取 R-CNN 源码：

```
$ git clone https://github.com/rbgirshick/rcnn.git
```

添加 Caffe v0.999 到 R-CNN 工程：

```
$ cdrcnn
$ ln -sf $CAFFE_ROOT/external/caffe
```

下载预先训练过的模型:

```
$ ./data/fetch_models.sh
```

```
Downloading precomputed R-CNN models (1.5G)...
```

```
--2015-10-26 23:54:36-- http://www.cs.berkeley.edu/~rbg/r-cnn-release1-data.tgz
```

```
Resolving www.cs.berkeley.edu (www.cs.berkeley.edu)... 128.32.244.183
```

```
Connecting to www.cs.berkeley.edu (www.cs.berkeley.edu)|128.32.244.183|:80...
connected.
```

```
HTTP request sent, awaiting response... 200 OK
```

```
Length: 1539453962 (1.4G) [application/x-tar]
```

```
Saving to: 'r-cnn-release1-data.tgz'
```

```
100%[=====>] 1,539,453,962 632KB/s in 12m 45s
```

```
2015-10-27 00:07:22 (1.92 MB/s) - 'r-cnn-release1-data.tgz' saved [1539453962/1539453962]
```

```
Unzipping...
```

```
caffe_nets/
```

```
caffe_nets/finetune_voc_2007_trainval_iter_70k
```

```
caffe_nets/finetune_voc_2012_train_iter_70k
```

```
caffe_nets/ilsvrc_2012_train_iter_310k
```

```
caffe_nets/finetune_ilsvrc13_val1+train1k_iter_50000
```

```
rcnn_models/
```

```
rcnn_models/voc_2012/
```

```
rcnn_models/voc_2012/rcnn_model_finetuned.mat
```

```
rcnn_models/voc_2012/bbox_regressor_final.mat
```

```
rcnn_models/voc_2007/
```

```
rcnn_models/voc_2007/rcnn_model_finetuned.mat
```

```
rcnn_models/voc_2007/bbox_regressor_final.mat
```

```
rcnn_models/ilsvrc2013/
```

```
rcnn_models/ilsvrc2013/rcnn_model.mat
```

```
rcnn_models/ilsvrc2013/bbox_regressor_final.mat
```

```
Done. Please run this command again to verify that checksum =
758aff9fa51d830be3281f91a8b03126.
```

```
$ ./data/fetch_selective_search_data.sh
```

```
Downloading precomputed selective search boxes (1.8G)...
```

```
--2015-10-27
```

```
12:49:18--
```

```
http://www.cs.berkeley.edu/~rbg/r-cnn-release1-selective-search.tgz
```

```
Resolving www.cs.berkeley.edu (www.cs.berkeley.edu)... 128.32.244.183
```

```
Connecting to www.cs.berkeley.edu (www.cs.berkeley.edu)|128.32.244.183|:80...
```

```

connected.
HTTP request sent, awaiting response... 200 OK
Length: 1931878932 (1.8G) [application/x-tar]
Saving to: â€˜r-cnn-release1-selective-search.tgzâ€™

100%[=====>] 1,931,878,932 119KB/s in 4h 48m

2015-10-27 17:37:45 (109 KB/s) - â€˜r-cnn-release1-selective-search.tgzâ€™ saved
[1931878932/1931878932]

Unzipping...
selective_search_data/
selective_search_data/voc_2012_test.mat
selective_search_data/voc_2007_trainval.mat
selective_search_data/voc_2007_test.mat
selective_search_data/voc_2012_val.mat
selective_search_data/voc_2012_train.mat
selective_search_data/voc_2007_val.mat
selective_search_data/voc_2007_train.mat
selective_search_data/voc_2012_trainval.mat
selective_search_data/ilsvrc13_test.mat
selective_search_data/ilsvrc13_val1.mat
selective_search_data/ilsvrc13_val2.mat
selective_search_data/ilsvrc13_val.mat
Done. Please run this command again to verify that checksum =
6cf6df219c1e514f64482f11d00bd0b4.

```

运行 Matlab:

```
$ matlab
```

如果提示下载 Selective Search 源码, 则按照提示下载。

当看到“R-CNN startup done”并弹出 Matlab 命令提示符“>>”时, 在 Matlab 中运行:

```
>>rcnn_build()
```

这时将会编译 liblinear 和 Selective Search。如果编译器有警告, 可以忽略。

编译结束后, 验证 Caffe 及其 Matlab 包装设置没有问题:

```
>>key = caffe('get_init_key');
```

预期输出为 key = -2。

运行 Demo:

```
>>rcnn_demo
```

经过一段时间后（大约 40 多秒），得到输出如图 19-12 所示。



图19-12 R-CNN Demo输出

限于篇幅，本节不再深入。读者可以进一步阅读参考资料[4]及 R-CNN 工程代码，循序渐进地学习^{[5][6][7]}，挑战目标检测这个世界性难题，如果有机会可以刷刷 ILSVRC 比赛。

19.4 人脸识别

人脸识别技术的研究始于 20 世纪 60 年代，是近些年来计算机模式识别领域中一个非常活跃的研究课题，在安全验证系统、信用卡验证、医学信息、档案管理、视频会议、人机交互、系统公安（罪犯识别）等方面有着巨大的应用前景。

19.4.1 问题描述

通俗地讲，人脸识别就是解决“图片中的人是谁”的问题，它主要分为人脸验证（face verification）和人脸识别（face identification）两类，即通常所说的 1:1 和 1:n 的问题。

人脸识别领域最著名的数据集是 LFW（Labeled Faces in the Wild，带标签的自然人脸数据库）数据集^[8]，由美国马萨诸塞州大学艾姆赫斯特学院创建。

完整的带标签的自然人脸数据库可以作为一个压缩包下载，解压缩后，数据库内容会置于

新的目录 lfw 中。总共有 13233 张 JPEG 格式图片，属于 5749 个不同的人，每张图片尺寸都是 250×250 。

人脸识别技术面临很多挑战，比如：如何适应各种人脸姿态、各种表情、年龄变化、人种、性别、变化的光线、分辨率的差别、图像退化、遮挡（如口罩、墨镜、头发、胡须）等，传统的人脸识别技术很难有效地克服这些因素。近年来，深度学习在大规模图像分类问题上取得了长足的进步，基于深度学习的人脸识别研究不断刷新 LFW 排行榜，其中代表性工作主要有 FaceBook 的 DeepFace、香港中文大学汤晓鸥研究组的 DeepID2 以及国内的 FACE++ 团队。

人脸识别技术在学术界和工业界吸引了越来越多的研究者加入，而最新的 LFW 数据库测试结果显示，基于深度学习的人脸识别算法（1:1）已经达到 99.15% 的准确率。但是在 LFW 取得的指标超过人类的结果并不足以说明技术真正实用化，在一个特殊集合上训练得到的模型无法很好地推广到其他集合，人脸识别技术完全实用化还面临很多挑战。

一般来说，人脸识别系统包括图像获取、人脸定位、图像预处理以及人脸识别（身份确认或者身份查找），如图 19-13 所示。

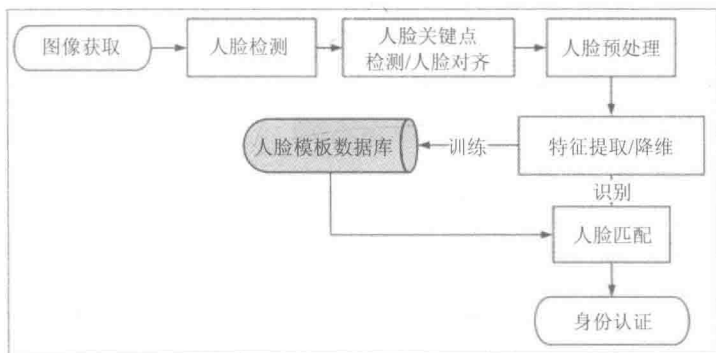


图19-13 人脸识别系统框架

系统输入一般是一张或者一系列含有未确定身份的人脸图像，以及人脸数据库中若干已知身份的人脸图像或者相应的人脸特征编码，而其输出则是一系列相似度分数，表明待识别人脸的身份。

1. 人脸检测

人脸检测主要用于定位图像中的人脸，如果一张图片有多个人，需要将他们全部检出。Haar+Adaboost 是一种比较经典的人脸检测框架，但其在实际复杂场景中的检测性能往往并不理想。

2. 关键点定位

关键点定位主要是在已知人脸所在位置的基础上，用于自动标注人脸的五官位置，如眼睛、鼻子、嘴巴等位置。传统的基于 ASM (Active Shape Model)、AAM (Active Appearance Model) 的关键点定位方法，并不能很好地处理大角度、复杂光照和夸张表情的情况。

3. 特征提取/降维

深度学习网络结构通过对海量的互联网人脸数据进行监督学习，提取出对姿态、光照等干扰因素鲁棒的人脸特征。一般来说，深度学习特征冗余度很高，需要对其做进一步的降维处理以获得更加紧凑的特征。传统的 PCA+LDA 方法，需要假设训练样本服从多元高斯分布，当实际样本分布条件不满足时，需要一些经验性的参数调优，不适合大规模人脸数据的训练。在实际中可采用基于 Pairwise 的 Metric Learning 降维方法，通过减少类内距离，扩大类间距离，效果会好一些。

19.4.2 最佳实践——使用 Face++ SDK 实现人脸检测

Face++ SDK 是国内最早推出的人脸识别公共服务。本节介绍如何利用该 SDK 实现人脸关键点检测。

操作步骤如下：

- 注册用户 (<http://www.faceplusplus.com/>)。
- 在开发者中心 (DevCenter) 创建应用并获取 API_KEY、API_SECRET。
- 运行 Matlab，下载 SDK (<https://github.com/FacePlusPlus/facepp-matlab-sdk>)。
- 更改 facepp_demo.m 中 API_KEY 和 API_SECRET 为注册的信息。
- 按 F5 键运行，得到结果如图 19-14 所示。



图19-14 Face++ SDK人脸检测Demo

19.5 自然语言处理

自然语言处理（Natural Language Processing, NLP）是涵盖计算机科学、人工智能和语言学的交叉学科，目的是让计算机处理、理解自然语言，同人类进行更直接的交互。

19.5.1 问题描述

自然语言处理包括词法分析、句法分析、语义分析等多个层次，如图 19-15 所示。

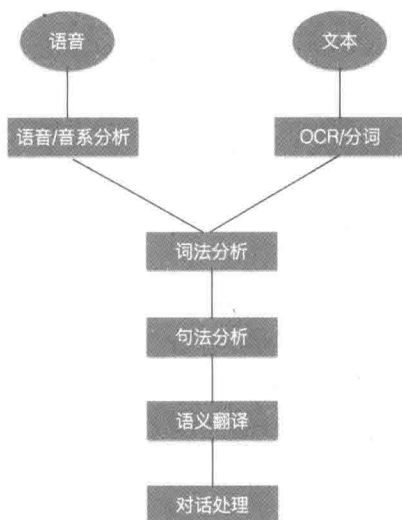


图19-15 NLP的层次

深度学习最早在语音识别（2009 年^[8]）和计算机视觉（2012 年^[9]）领域大放异彩，最近才应用到自然语言处理领域（称为 Deep NLP），仍有很多挑战性的工作，是当前的研究热点。

Deep NLP 将 NLP 的思路和目标与表示学习、深度学习方法结合。

- 在语音/音系分析层次，深度学习可以通过声音特征直接预测音素、单词，并表示为向量。
- 在词法分析层次，每个语素表示为向量，神经网络将两个向量合并为一个向量。
- 在句法分析层次，每个单词或短语表示为向量，神经网络将两个向量合并为一个向量。
- 在语义分析层次，每个单词、短语、逻辑表达都表示为向量，神经网络将两个向量合并为一个向量。

典型的 NLP 应用从简单到复杂有：拼写检查、关键词搜索、同义词发现、网络爬虫、文本分类、自动文摘、文本校对/纠错、机器翻译、复杂对话系统等。目前 NLP 的难点在于语言表达具有多样性、上下文相关性、模糊性。

19.5.2 最佳实践——NLP-Caffe

本节使用斯坦福大学 NLP 研究组 (<http://nlp.stanford.edu/>) 开发的 NLP-Caffe 进行实践，源码可以从 GitHub 上获取：<https://github.com/Russell91/nlpcaffe>。

```
$git clone https://github.com/Russell91/nlpcaffe.git
Cloning into 'nlpcaffe'...
remote: Counting objects: 21028, done.
remote: Total 21028 (delta 0), reused 0 (delta 0), pack-reused 21028
Receiving objects: 100% (21028/21028), 33.16 MiB | 5.47 MiB/s, done.
Resolving deltas: 100% (13810/13810), done.
Checking connectivity... done.
$cdnlpcaffe/
$ls
caffe.cloc CONTRIBUTORS.md  includematlabsrc
CHANGES.txt              data                      INSTALL.md                models                    tools
cmakedocker               LICENSE                  python
CMakeLists.txt            docs                     Makefile                  README.md
CONTRIBUTING.md          examples Makefile.config.example scripts
```

可以看出，NLP-Caffe 源码与 Caffe 几乎完全一致。实际上 NLP-Caffe 是 Caffe 的一个分支，NLP 用户无须触碰 C++ 代码就可以直接上手使用。为了编译 NLP-Caffe，我们将前面编译好的 Caffe 目录下的 Makefile.config 拷贝一份，放于 NLP-Caffe 目录下，直接编译即可：

```
$cp$CAFFE_ROOT/Makefile.config .
$ make -j
PROTOC src/caffe/proto/caffe.proto
CXX src/caffe/parallel.cpp
CXX src/caffe/syncedmem.cpp
CXX src/caffe/internal_thread.cpp
CXX src/caffe/data_reader.cpp
CXX src/caffe/util/db_leveldb.cpp
CXX src/caffe/util/insert_splits.cpp
CXX src/caffe/util/cudnn.cpp
CXX src/caffe/util/im2col.cpp
```



```
CXX src/caffe/util/db_lmdb.cpp
CXX src/caffe/util/upgrade_proto.cpp
CXX src/caffe/util/math_functions.cpp
CXX src/caffe/util/io.cpp
CXX src/caffe/util/hdf5.cpp
CXX src/caffe/util/db.cpp
CXX src/caffe/util/signal_handler.cpp
CXX src/caffe/util/benchmark.cpp
CXX src/caffe/util/blocking_queue.cpp
CXX src/caffe/data_transformer.cpp
CXX src/caffe/solvers/adam_solver.cpp
CXX src/caffe/solvers/adadelta_solver.cpp
CXX src/caffe/solvers/rmsprop_solver.cpp
CXX src/caffe/solvers/nesterov_solver.cpp
CXX src/caffe/solvers/sgd_solver.cpp
CXX src/caffe/solvers/adagrad_solver.cpp
CXX src/caffe/layer.cpp
CXX src/caffe/solver.cpp
CXX src/caffe/layers/input_layer.cpp
CXX src/caffe/layers/cudnn_lrn_layer.cpp
CXX src/caffe/layers/filter_layer.cpp
CXX src/caffe/layers/slice_layer.cpp
CXX src/caffe/layers/memory_data_layer.cpp
CXX src/caffe/layers/concat_layer.cpp
CXX src/caffe/layers/softmax_layer.cpp
CXX src/caffe/layers/contrastive_loss_layer.cpp
CXX src/caffe/layers/mvn_layer.cpp
CXX src/caffe/layers/reduction_layer.cpp
CXX src/caffe/layers/pooling_layer.cpp
CXX src/caffe/layers/reshape_layer.cpp
CXX src/caffe/layers/batch_reindex_layer.cpp
CXX src/caffe/layers/cudnn_conv_layer.cpp
CXX src/caffe/layers/cudnn_pooling_layer.cpp
CXX src/caffe/layers/data_layer.cpp
CXX src/caffe/layers/threshold_layer.cpp
CXX src/caffe/layers/log_layer.cpp
CXX src/caffe/layers/lstm_layer.cpp
CXX src/caffe/layers/silence_layer.cpp
CXX src/caffe/layers/exp_layer.cpp
CXX src/caffe/layers/window_data_layer.cpp
CXX src/caffe/layers/bnll_layer.cpp
```

```
CXX src/caffe/layers/dropout_layer.cpp
CXX src/caffe/layers/cudnn_softmax_layer.cpp
CXX src/caffe/layers/tanh_layer.cpp
CXX src/caffe/layers/cudnn_lcn_layer.cpp
CXX src/caffe/layers/cudnn_relu_layer.cpp
CXX src/caffe/layers/elu_layer.cpp
CXX src/caffe/layers/split_layer.cpp
CXX src/caffe/layers/hdf5_output_layer.cpp
CXX src/caffe/layers/flatten_layer.cpp
CXX src/caffe/layers/softmax_loss_layer.cpp
CXX src/caffe/layers/argmax_layer.cpp
CXX src/caffe/layers/hdf5_data_layer.cpp
CXX src/caffe/layers/spp_layer.cpp
CXX src/caffe/layers/prelu_layer.cpp
CXX src/caffe/layers/deconv_layer.cpp
CXX src/caffe/layers/relu_layer.cpp
CXX src/caffe/layers/crop_layer.cpp
CXX src/caffe/layers/bias_layer.cpp
CXX src/caffe/layers/hinge_loss_layer.cpp
CXX src/caffe/layers/neuron_layer.cpp
CXX src/caffe/layers/sigmoid_cross_entropy_loss_layer.cpp
CXX src/caffe/layers/dummy_data_layer.cpp
CXX src/caffe/layers/multinomial_logistic_loss_layer.cpp
CXX src/caffe/layers/base_conv_layer.cpp
CXX src/caffe/layers/embed_layer.cpp
CXX src/caffe/layers/accuracy_layer.cpp
CXX src/caffe/layers/wordvec_layer.cpp
CXX src/caffe/layers/sigmoid_layer.cpp
CXX src/caffe/layers/eltwise_layer.cpp
CXX src/caffe/layers/cudnn_sigmoid_layer.cpp
CXX src/caffe/layers/power_layer.cpp
CXX src/caffe/layers/inner_product_layer.cpp
CXX src/caffe/layers/lrn_layer.cpp
CXX src/caffe/layers/loss_layer.cpp
CXX src/caffe/layers/scale_layer.cpp
CXX src/caffe/layers/absval_layer.cpp
CXX src/caffe/layers/infogain_loss_layer.cpp
CXX src/caffe/layers/euclidean_loss_layer.cpp
CXX src/caffe/layers/image_data_layer.cpp
CXX src/caffe/layers/tile_layer.cpp
CXX src/caffe/layers/base_data_layer.cpp
```

```
CXX src/caffe/layers/conv_layer.cpp
CXX src/caffe/layers/cudnn_tanh_layer.cpp
CXX src/caffe/layers/batch_norm_layer.cpp
CXX src/caffe/layers/im2col_layer.cpp
CXX src/caffe/net.cpp
CXX src/caffe/common.cpp
CXX src/caffe/blob.cpp
CXX src/caffe/layer_factory.cpp
CXX tools/extract_features.cpp
CXX tools/train_net.cpp
CXX tools/caffe.cpp
CXX tools/test_net.cpp
CXX tools/compute_image_mean.cpp
CXX tools/upgrade_solver_proto_text.cpp
CXX tools/net_speed_benchmark.cpp
CXX tools/upgrade_net_proto_text.cpp
CXX tools/device_query.cpp
CXX tools/convert_imageset.cpp
CXX tools/finetune_net.cpp
CXX tools/upgrade_net_proto_binary.cpp
CXX examples/cifar10/convert_cifar_data.cpp
CXX examples/siamese/convert_mnist_siamese_data.cpp
CXX examples/cpp_classification/classification.cpp
CXX examples/mnist/convert_mnist_data.cpp
CXX .build_release/src/caffe/proto/caffe.pb.cc
AR -o .build_release/lib/libcaffe.a
LD -o .build_release/lib/libcaffe.so.1.0.0-rc3
CXX/LD -o .build_release/tools/train_net.bin
CXX/LD -o .build_release/tools/extract_features.bin
CXX/LD -o .build_release/tools/caffe.bin
CXX/LD -o .build_release/tools/test_net.bin
CXX/LD -o .build_release/tools/compute_image_mean.bin
CXX/LD -o .build_release/tools/upgrade_solver_proto_text.bin
CXX/LD -o .build_release/tools/net_speed_benchmark.bin
CXX/LD -o .build_release/tools/upgrade_net_proto_text.bin
CXX/LD -o .build_release/tools/device_query.bin
CXX/LD -o .build_release/tools/convert_imageset.bin
CXX/LD -o .build_release/tools/finetune_net.bin
CXX/LD -o .build_release/tools/upgrade_net_proto_binary.bin
CXX/LD -o .build_release/examples/cifar10/convert_cifar_data.bin
CXX/LD -o .build_release/examples/siamese/convert_mnist_siamese_data.bin
```

```
CXX/LD -o .build_release/examples/cpp_classification/classification.bin
CXX/LD -o .build_release/examples/mnist/convert_mnist_data.bin
```

接下来还要编译 pycaffe (Python 依赖包有准备过程参考 16.2.1 节):

```
# make pycaffe
CXX/LD -o python/caffe/_caffe.so python/caffe/_caffe.cpp
touch python/caffe/proto/__init__.py
PROTOC (python) src/caffe/proto/caffe.proto
```

安装额外的 Python 运行时依赖包 py-lmdb, 这个包在原版 Caffe 中不是必需的:

```
$ sudo pip install lmdb
Downloading/unpacking lmdb
  Downloading lmdb-0.89.tar.gz (149kB): 149kB downloaded
  Running setup.py (path:/tmp/pip_build_root/lmdb/setup.py) egg_info for package lmdb
    py-lmdb: Using bundled liblmdb; override with LMDB_FORCE_SYSTEM=1.
    py-lmdb: Using CPython extension; override with LMDB_FORCE_CFFI=1.

    warning: no directories found matching '.'
Installing collected packages: lmdb
  Running setup.py install for lmdb
    py-lmdb: Using bundled liblmdb; override with LMDB_FORCE_SYSTEM=1.
    py-lmdb: Using CPython extension; override with LMDB_FORCE_CFFI=1.
    building 'cpython' extension
    x86_64-linux-gnu-gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall
-Wstrict-prototypes -fPIC -Ilib/py-lmdb -Ilib -I/usr/include/python2.7 -c lmdb/cpython.c
-o build/temp.linux-x86_64-2.7/lmdb/cpython.o -UNDEBUG -w
    x86_64-linux-gnu-gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall
-Wstrict-prototypes -fPIC -Ilib/py-lmdb -Ilib -I/usr/include/python2.7 -c lib/mdb.c -o
build/temp.linux-x86_64-2.7/lib/mdb.o -UNDEBUG -w
    x86_64-linux-gnu-gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall
-Wstrict-prototypes -fPIC -Ilib/py-lmdb -Ilib -I/usr/include/python2.7 -c lib/midl.c -o
build/temp.linux-x86_64-2.7/lib/midl.o -UNDEBUG -w
    x86_64-linux-gnu-gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions
-Wl,-Bsymbolic-functions -Wl,-z,relro -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall
-Wstrict-prototypes -D_FORTIFY_SOURCE=2 -g -fstack-protector --param=ssp-buffer-size=4
-Wformat -Werror=format-security build/temp.linux-x86_64-2.7/lmdb/cpython.o
build/temp.linux-x86_64-2.7/lib/mdb.o build/temp.linux-x86_64-2.7/lib/midl.o -o
build/lib.linux-x86_64-2.7/lmdb/cpython.so

    warning: no directories found matching '.'
Successfully installed lmdb
Cleaning up...
```

准备数据:

```

$ ./data/language_model/get_lm.sh Downloading...
--2016-05-05 06:57:55-- http://russellsstewart.com/s/lm/vocab.pkl
Resolving russellsstewart.com (russellsstewart.com)... 52.8.97.128
Connecting to russellsstewart.com (russellsstewart.com)|52.8.97.128|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1693706 (1.6M) [application/octet-stream]
Saving to: 'vocab.pkl'

100%[=====>] 1,693,706 1.52MB/s in 1.1s

2016-05-05 06:58:00 (1.52 MB/s) - 'vocab.pkl' saved [1693706/1693706]

--2016-05-05 06:58:00-- http://russellsstewart.com/s/lm/train_indices.txt
Resolving russellsstewart.com (russellsstewart.com)... 52.8.97.128
Connecting to russellsstewart.com (russellsstewart.com)|52.8.97.128|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 28632225 (27M) [text/plain]
Saving to: 'train_indices.txt'

100%[=====>] 28,632,225 5.32MB/s in 6.5s

2016-05-05 06:58:07 (4.19 MB/s) - 'train_indices.txt' saved [28632225/28632225]

--2016-05-05 06:58:07-- http://russellsstewart.com/s/lm/valid_indices.txt
Resolving russellsstewart.com (russellsstewart.com)... 52.8.97.128
Connecting to russellsstewart.com (russellsstewart.com)|52.8.97.128|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 278599 (272K) [text/plain]
Saving to: 'valid_indices.txt'

100%[=====>] 278,599 368KB/s in 0.7s

2016-05-05 06:58:08 (368 KB/s) - 'valid_indices.txt' saved [278599/278599]

--2016-05-05 06:58:08-- http://russellsstewart.com/s/lm/test_indices.txt
Resolving russellsstewart.com (russellsstewart.com)... 52.8.97.128
Connecting to russellsstewart.com (russellsstewart.com)|52.8.97.128|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 279763 (273K) [text/plain]

```

```

Saving to: 'test_indices.txt'

100%[=====>] 279,763      393KB/s   in 0.7s

2016-05-05 06:58:09 (393 KB/s) - 'test_indices.txt' saved [279763/279763]

Done.

```

使用刚刚下载的数据, 运行如下命令可以生成 LMDB 数据库和模型结构描述文件 `train_val.prototxt`:

```

$ python ./examples/language_model/create_lm.py --make_data
Starting train
Writing 306068 sentences
Starting valid
Writing 3000 sentences
Starting test
Writing 3000 sentences

```

接着运行训练:

```
$ ./examples/language_model/train_lm.sh
```

输出很长, 不再贴出。感兴趣的读者可以自行试验, 并按照本书前面的方法深入研究数据和模型以及训练过程。

TIPS: 用可视化方法研究可以更快地获得灵感。

通过本节内容, 读者初步认识了自然语言处理问题和实现方法, 进一步学习可以参考资料^[10]。

19.6 艺术风格^{[11][12]}

19.6.1 问题描述

在好的艺术作品中, 尤其是油画中, 人类掌握了通过在内容和风格上实施复杂的重复创建独特视觉体验的技能。谁也不知道该过程的算法基础, 也不存在具有相似特性的人工系统。然而, 在其他视觉感知关键领域, 例如目标和人脸识别中, 神经网络展示了接近人类的性能。

本节我们介绍基于深度神经网络的人工系统, 创建高质量艺术图片。该系统使用神经表示来分离、重组任意图片的内容和风格, 提供了用于创建艺术图片的神经算法。更多的, 在性能

优化的人工神经网络和生物视觉系统之间奇特的相似性，提供了一种理解人类如何创建和感知艺术想象的途径。

1. 如何获得输入图像的风格表示

使用基于网络的每层不同滤波器响应互相关特征空间（最初设计为提取纹理信息）。通过多个层特征互相关可以提取输入图像的稳定、多尺度纹理信息，即为其图像风格表示，效果如图 19-16 上半部分所示。

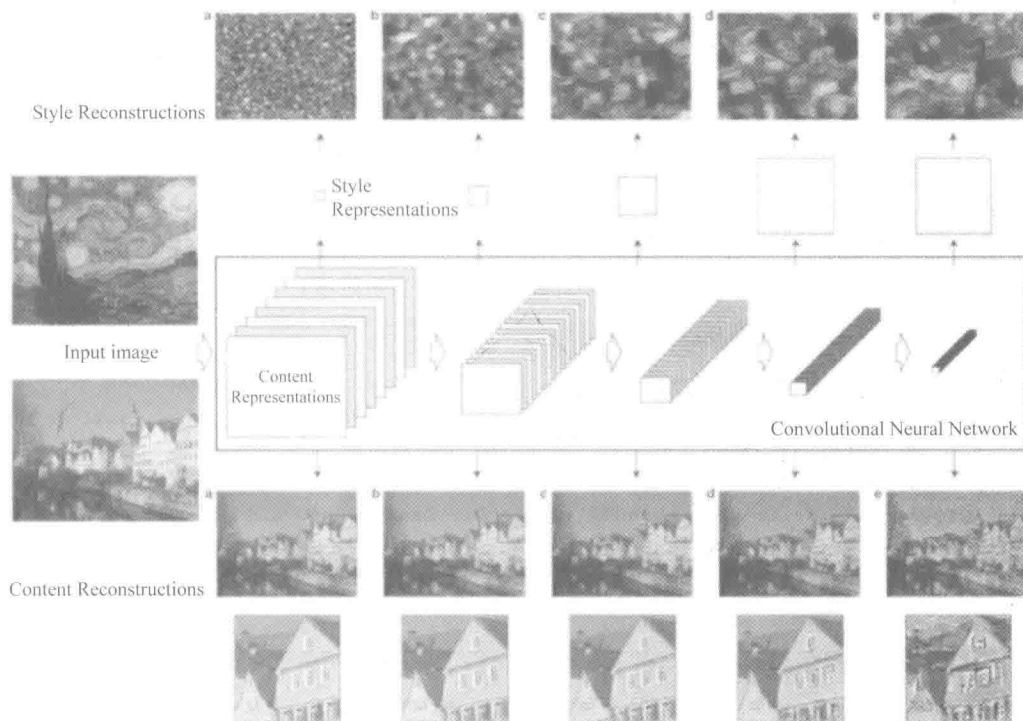


图19-16 风格表示与内容表示

2. 如何获得输入图像的内容表示

将各层重构原图，效果如图 19-16 下半部分所示。可以发现随着重构层的加深，损失了图像细节信息，但保留了主题信息，故高层特征保留了图像的内容表示。

发现图像的风格表示和内容表示具有可分离性，这就为艺术风格转换提供了土壤。一种基

于随机梯度下降搜索的方法自然而然生。从一张白噪声图像 C0 开始，寻找同时满足两个条件的图像：

- 与图像 A 的内容表示相匹配。
- 与艺术作品 B 的风格表示相匹配。

最终结果 C 既具有 A 的整体布局信息，又具有 B 的颜色和局部结构信息。

19.6.2 最佳实践——style-transfer

本节使用 Caffe 实现图像艺术风格转换。代码来自：<https://github.com/fzliu/style-transfer.git>，是基于 pycaffe 对参考资料[11]中方法的实现。其中神经网络计算由 Caffe 负责完成，而损失最小化、其他杂七杂八的矩阵操作使用 numpy 和 scipy 完成，这里最优化方法采用 L-BFGS。

读者在进行本节试验前，首先需要准备一份编译好的 Caffe 并编译 pycaffe，具体步骤请参考 16.2.1 节。此外需要安装 Python progressbar 包，用于显示进度：

```
$ sudo pip install progressbar
```

```
$ git clone https://github.com/fzliu/style-transfer.git
```

```
Cloning into 'style-transfer'...
```

```
remote: Counting objects: 354, done.
```

```
remote: Total 354 (delta 0), reused 0 (delta 0), pack-reused 354
```

```
Receiving objects: 100% (354/354), 1.71 MiB | 15.00 KiB/s, done.
```

```
Resolving deltas: 100% (188/188), done.
```

```
Checking connectivity... done.
```

```
$ cd style-transfer/
```

```
$ tree
```

```
.
├── demo.py
├── images
│   ├── content
│   │   ├── johannesburg.jpg
│   │   ├── nanjing.jpg
│   │   └── sanfrancisco.jpg
│   └── results
│       ├── starry_johannesburg.jpg
│       ├── starry_nanjing.jpg
│       └── starry_sanfrancisco.jpg
```



```

|   └── style
|       └── starry_night.jpg
├── models
|   ├── caffeNet
|   |   ├── deploy.prototxt
|   |   └── ilsvrc_2012_mean.npy
|   ├── googlenet
|   |   ├── deploy.prototxt
|   |   └── ilsvrc_2012_mean.npy
|   ├── stylenet
|   |   └── ilsvrc_2012_mean.npy
|   ├── vgg16
|   |   ├── ilsvrc_2012_mean.npy
|   |   └── VGG_ILSVRC_16_layers_deploy.prototxt
|   └── vgg19
|       ├── ilsvrc_2012_mean.npy
|       └── VGG_ILSVRC_19_layers_deploy.prototxt
├── README.md
├── requirements.txt
├── scripts
|   └── download_models.sh  // 下载模型
└── style.py                // 所有核心代码都在这里

```

11 directories, 21 files

使用如下命令下载所需的模型:

```
$ ./scripts/download_models.sh
```

一切准备就绪:

```
$ python style.py -s <style_image> -c <content_image> -m <model_name> -g 0
```

其中, “-s <style_image>” 指定了风格图片; “-c <content_image>” 指定了内容图片; “-m <model_name>” 指定了用什么模型; “-g 0” 指定了使用哪块 GPU, 如果设为-1, 则表示在 CPU 上运行。欲了解更详细的命令行参数, 请阅读源码。

这里运行参数设置如下:

```
$ python style.py -s images/style/starry_night.jpg \
-c images/content/nanjing.jpg \
-g 0
```

得到结果如图 19-17 所示。



内容图片

风格图片

转换结果

图19-17 图像风格转换结果

19.7 小结

今天我们跟上了时代步伐，领略了深度学习在图形图像、文本、视频、艺术作品等领域的进展。值得读者深思的是，将深度学习大法与传统领域结合可以迸发无限可能，只有你想不到的。

忽悠是一拍脑袋就想出来，然后就忘了。

创新是一拍脑袋就想出来，然后实现了。

19.8 练习题

1. 搜集深度学习在如下领域中的应用情况：场景识别，动作、行为识别，医学研究，天文观测，海洋气象，地震信号处理。

2. 试着将左图转换为右图风格：



19.9 参考资料

- [1] 斯坦福大学公开课 CS231n, Convolutional Neural Networks for Visual Recognition
- [2] 阿里巴巴淘宝技术部 ATA 文章. 通平. 深度学习及其在淘宝图像应用探讨. 2014.6
- [3] 阿里巴巴安全部 ATA 文章. 觉奥. 身份认证那些事儿 (二): 实人认证中的 OCR 技术. 2015.11
- [4] R. Girshick et al. Rich feature hierarchies for accurate object detection and semantic segmentation, CVPR, 2014.
- [5] R. Girshick, "Fast r-cnn", ICCV, 2015.
- [6] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," NIPS, 2015
- [7] T-CNN: Tubelets with Convolutional Neural Networks for Object Detection from Videos, arXiv:1604.02532v1
- [8] Geoffrey Hinton, Deep Belief Networks for phone recognition
- [9] Alex Krizhevsky, Geoffrey Hinton, Image Net Classification with deep convolutional Neural Networks
- [10] 斯坦福大学公开课 CS224d, Deep Learning for Natural Language Processing, Richard Socher
- [11] A Neural Algorithm of Artistic Style, arXiv:1508.06576v2
- [12] Exploring the Neural Algorithm of Artistic Style, arXiv:1602.07188v1

第 20 天

继往开来的领路人

本书即将结尾，读者前进的路却仍然伸向远方。

在前进时，不可避免会遇到荆棘丛生、泥泞坎坷的险途。

今天的内容是为读者提供向导，指引方向，避免深陷泥潭。

20.1 Caffe Traps and Pitfalls

本节是来给 Caffe 挑刺的。每种框架实现时都会关注某些方面，而忽视了另一些方面，所以没有十全十美的框架，只有更适合自己需求的框架。当使用时如果能提前知道这些框架的坑和缺点，就能未雨绸缪，有备无患。

20.1.1 不支持任意数据类型

Caffe 大部分数据结构（如 Blob、Layer、Net、Solver 及其派生类）都是模板类，通过实例化模板参数来支持多种数据类型。

事实上，Caffe 只支持 float 和 double 两种类型网络。如果创建 Blob<int>实例，则会发现缺少 Update() 计算实现。Caffe 依赖 BLAS 库实现基本运算，BLAS 不支持的数据类型，Caffe 同样不支持。

Caffe 创建网络时一旦使用某种类型，整个网络中各层就都是相同类型（通过模板参数传递，参考 11.2.1 节中的 Net::Init() 函数实现），限制了同一网络多种不同数据类型支持的需求。如果读者希望使用低精度数值类型实现某些层计算（ReLU 和 Pooling 对数值精度要求不高），则需

要修改 Caffe 中相应代码，打破模板参数的默认传递路径。

20.1.2 不够灵活的高级接口

Caffe 支持 3 种高级接口：命令行、Python 和 Matlab，三者在功能上几乎雷同，定义一个模型仍然需要用 ProtoBuf 文件描述，然后调用 Net 构造函数，将 ProtoBuf 描述文件名作为参数传递给 Net 进行网络创建。也就是说，无论哪种高级接口都不能逃避手写（或者工具写）ProtoBuf 网络描述文件的工作。

ProtoBuf 相当于一门模型定义语言，虽然很强大，但初学者写起来还是需要一定量的积累才能掌握。不深入阅读 Caffe 代码，可能永远无法理解 ProtoBuf 描述文件中的一些参数、集合的规则。

20.1.3 繁杂的依赖包

也许读者还记得第 5 天介绍的那些从没听过的一大堆依赖包安装过程；

也许读者还记得在某个暗无天日的夜晚辛苦配置了环境，结果一运行 `make` 就会弹出无数 Error 的噩梦。

Caffe 依赖包是出了名的繁杂。虽然使用现成轮子在实现上可以带来一定的便利性，但对于不熟悉这些依赖包的同学可能会引入新的学习难点。Caffe 大量的依赖包使得移植到嵌入式平台（小车、飞行器）变得困难且不经济。我曾将 Caffe 层层外表剥离，只保留最小的必需的代码，运行一个完整的 AlexNet 模型只需不到 500 行 C++ 代码。这部分代码可以轻松移植到除 x86 外的其他平台（GPU、ARM、DSP、Power8、FPGA 等）。

20.1.4 堪忧的卷积层实现

本书对于 Caffe 卷积层实现过程只字未提，主要是不希望读者花费大量时间和精力去学习一个速度慢而且实现极不优雅的方法。

凭什么说 Caffe 卷积层速度慢？有证据在此^[1]。读者细心观察会发现，Caffe 的（native 版）卷积层效率几乎无一例外都排名倒数。那为什么还有那么多人用 Caffe 呢？若不是及时支持 cuDNN，Caffe 早就 out 了。

Caffe 诞生时标榜自己“高效率”，利用 MKL、OpenBLAS、cuBLAS 等线性代数库，非常

“懒”地实现了“高效率”。代价是计算时需要 im2col 计算占用大量的临时空间，不适合内存紧俏的嵌入式平台和低端 GPU。

PS:Caffe 并不是完美支持 cuDNN,最新的 cuDNN v5 仍然未加入支持。体验 Caffe + cuDNN v5 的同学可能需要自己动手了，请参阅参考资料[2]。

20.1.5 架构之殇

Caffe 在设计之初看上去非常出众，宣传口号是改网络架构时不需要改哪怕一行代码！更不需要重新编译代码！

但随着时代的进步，这个口号逐渐变得“鸡肋”。深度学习新网络架构、新方法层出不穷，Caffe 在支持这些新特性时由于历史包袱，动作变得异常迟缓。不利因素^[3]如下：

- 使用 C++ 设计并实现新的 Layer，而模型定义仍需要用 ProtoBuf 描述，二者必须手动实现匹配。
- 新增一个层，需要手动实现 forward、backward、gradient update 三种算法。
- 为了支持 GPU，需要再手动实现一遍 GPU 版 forward、backward、gradient update。
- 新增层需要添加 proto 描述，如果添加了 caffe.proto 中的新层 id，那么很可能与其他分支冲突。
- 只支持单机多卡并行计算，不支持多机多卡分布式计算。
- 只支持数据级别并行，不支持模型级别并行。

20.1.6 应用场景局限性

Caffe 从一开始就主要面向计算机视觉、图像分类和识别、目标检测等领域，只考虑图像数据作为输入，而对语音、文本数据支持不好。这也导致使用 Caffe 的用户几乎无一例外都是用于图像类应用。

官方正在开发的 NLP-Caffe 是一个不错的尝试。参考资料[4]将语音信号提取的二维声谱图作为 Caffe 输入数据，进而实现声乐识别应用。这也是一个很好的解决问题的思路，值得读者学习。

20.2 最佳实践——Caffe2

Caffe 原作者贾扬清试验性重构了 Caffe，开启了独立分支，称为 Caffe2^[5]，Github 地址：
<https://github.com/Yangqing/caffe2>

Caffe2 出发点是改善 Caffe 的设计，使之更通用，不局限于计算机视觉领域，而是更广泛的机器学习任务。

相比 Caffe，Caffe2 在如下几个方面有了改进。

(1) 设备支持

从前是这样：

```
void Forward_cpu();
void Forward_gpu();
void Backward_cpu();
void Backward_gpu();
```

不够优雅，现在改为更简洁的接口：

```
template<typename Device> class Layer
{
    void Forward();
    void Backward()
}
```

(2) Blob

从前的 Caffe 是这样：

```
template<typename Dtype> class Blob
{
    Dtype* data; ...
};
```

现在改为：

```
class Blob {
    AnyPointer data;
    DataType type;
};
```

Blob 就是一切。放心，Blob 仍然是广义的 N 维数组。

下面我们来实际运行 Caffe2。有兴趣的读者可以按照本书传授的方法深入阅读 Caffe2 源码。

在安装成功 Caffe 的机器上再安装 Caffe2 简直易如反掌。所需依赖为 protobuf、glog、gflags、eigen3，其他依赖如 CUDA、cuDNN、OpenCV、OpenMPI、leveldb、lmdb、rocksdb、ZeroMQ 不是必需的。

下载源码：

```
$ git clone https://github.com/Yangqing/caffe2.git
$ cd caffe2
```

满足依赖后，直接编译：

```
$ make
```

运行 ipython notebook：

```
$ ./notebooks/start_ipython_notebook.sh
[I 09:48:53.482 NotebookApp] Writing notebook server cookie secret to
/run/user/0/jupyter/notebook_cookie_secret
[W 09:48:53.568 NotebookApp] WARNING: The notebook server is listening on all IP addresses
and not using encryption. This is not recommended.
[W 09:48:53.568 NotebookApp] WARNING: The notebook server is listening on all IP addresses
and not using authentication. This is highly insecure and not recommended.
[I 09:48:53.583 NotebookApp] Serving notebooks from local directory:
/disk1/deeplearning/caffe2
[I 09:48:53.583 NotebookApp] 0 active kernels
[I 09:48:53.583 NotebookApp] The IPython Notebook is running at: http://[all ip addresses
on your system]:8888/
[I 09:48:53.583 NotebookApp] Use Control-C to stop this server and shut down all kernels
(twice to skip confirmation).
[W 09:48:53.583 NotebookApp] No web browser found: could not locate runnable browser.
^^[I 09:50:22.176 NotebookApp] 302 GET / (10.168.154.40) 0.55ms
[I 09:50:59.192 NotebookApp] Writing notebook-signing key to /root/.local/share/
jupyter/notebook_secret
[W 09:50:59.194 NotebookApp] Notebook notebooks/alexnet2.ipynb is not trusted
[I 09:51:00.484 NotebookApp] Kernel started: 25cdda41-fc39-4dff-8d86-1a06c0dcfea6
```

打开浏览器，在地址栏输入：<http://127.0.0.1:8888/>，如图 20-1 所示。

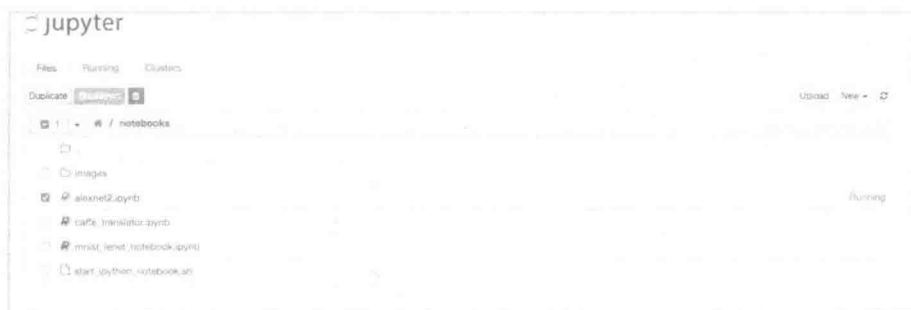


图20-1 IPython Notebook

点击 alexnet2.ipynb，进入 alexnet2 页面，如图 20-2 所示。

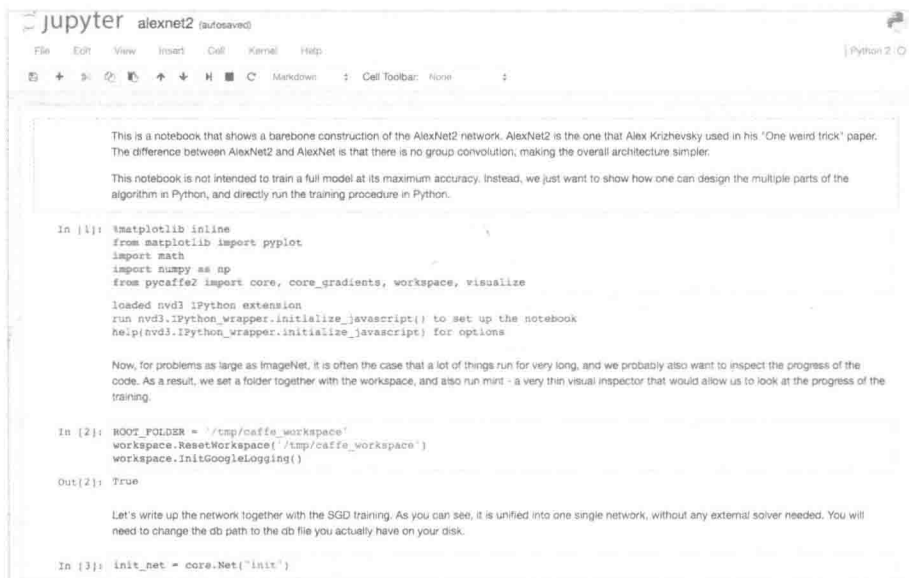


图20-2 alexnet2例程

更多关于 IPython Notebook 的使用，请参阅参考资料[6]。

20.3 练习题

1. 尽情吐槽或听取他人吐槽 Caffe。
2. 根据你的需求重构 Caffe。

20.4 参考资料

- [1] 卷积网络性能对比, <https://github.com/soumith/convnet-benchmarks>
- [2] Caffe + cuDNN V5, <http://blog.csdn.net/kkk584520/article/details/51163564>
- [3] 主流深度学习工具评估, <https://github.com/zer0n/deepframeworks>
- [4] Caffe 深度学习薛开宇笔记实例——基于卷积神经网络的声音识别
- [5] 贾扬清 CVPR 2015, Improving Caffe: Some Refactoring
- [6] IPython Notebook 简介 1, <http://hyry.dip.jp/tech/slice/slice.html/35>

第 21 天

新生

经过前面 20 天的刻苦努力，恭喜你马上就要毕业了！你可以亲自拿 Caffe 作为贴身武器，去解决所遇到的实际问题了！

21.1 三人行，必有我师

你不是一个人在战斗。

你身后还有庞大的 Caffe 社区。

Caffe 社区非常活跃，你遇到的所有问题几乎都有人遇到过，翻翻历史就能找到解决方案。建议你时常关注以下几个网址。

- Caffe 用户组: <https://groups.google.com/forum/#!forum/caffe-users>
- Caffe 讨论群: <https://gitter.im/BVLC/caffe>
- Caffe 问题反馈: <https://github.com/BVLC/caffe/issues>
- Caffe 中文社区: <http://www.caffecn.cn/>
- 微博: http://weibo.com/u/5847253963?is_hot=1
- CaffeCN QQ 群: 431141753

在与人交流的过程中，无论是支持还是反对，都是促进个人形成独立意识形态的过程，好的想法得到认可，错的想法得到纠正，这与机器学习过程相似。

21.2 路漫漫其修远兮，吾将上下而求索

本书只是入门读物，更进一步的修行需要靠读者在实际中不断总结经验教训。下面结出几个方向供读者参考。

(1) 跟进 Caffe

截至本书定稿时，Caffe 代码分支的变化如下：

- 支持 AMD GPU 及 OpenCL 开发环境：<https://github.com/bvlc/caffe/tree/opencl>
- 支持 Windows 平台（开发中）：<https://github.com/bvlc/caffe/tree/windows>
- 支持 Apache Spark 的独立分支：<https://github.com/yahoo/cafeonspark>

有条件的读者可以亲自搭环境测试一下。

(2) 跟进学术界研究动态

从一些影响力大的国外会议（如 ICCV、CVPR、NIPS、ECCV，以及期刊如 *IEEE Trans on Pattern Analysis and Machine Intelligence*、*IJCV*、*IEEE Trans on Image Processing*、*Pattern Recognition* 等）中获取计算机视觉、机器学习、深度学习相关的最新论文，跟进最前沿的研究动态。

(3) 阅读其他深度学习框架代码

推荐阅读并实际使用如下开源深度学习框架。

- 关注分布式训练：MxNet、CNTK、TensorFlow
- 关注 GPU 上性能优化：cuda-convnet2
- 关注高级语言灵活性：Torch、Theano

(4) 自己实现一个深度学习框架

精通的标准，应该是这样的：徒手写出 Caffe 每个模块；徒手写 prototxt 设计网络模型；前一个是针对工程师，后一个是针对学术研究人员。另外，对于多个深度学习框架，应做到自由

转换不同框架的模型描述。

深度学习的知识点是一张网，而经典教材会将内容组织为树状结构利于理解（恰如深度学习本身架构）。本书则组织为线，适合入门级读者。通读本书后，读者可以毫无压力地读国外大部头书籍，能够让自己的知识结构更加完备。而将知识与应用结合才是最终目的，所以读者最好根据自身情况选择适合自己的方向专攻。

篇尾语

下篇•升华主要内容是引领读者熟悉如何在实际生产和应用中使用 Caffè，了解企业一线员工的日常工作内容，为今后从事该行业做好心理和生理准备。

“你是否愿意和我不离不弃，共度一生？”——Caffè 如是说。

你需要包容它的缺点。

你需要时常关心和爱护它。

你可以让它变得更美、更优秀、更贴心。

结束语

余自幼好读书，史书尤甚，每得一佳作，便欣然忘食，沉浸其间，不能自己。观春秋之五霸，阅战国之纷争，诵秦时明月汉时关，叹三国，悲水浒，泣红楼。

经典之所以经典，是无论什么时候读，都会有不同的体验。

Caffe 代码也是常读常新，每次阅读仿佛都有新发现。除了作为理解深度学习原理的入门工具，更多的能从中学习一个完整工程的方方面面，以及不断优化的精益求精气质。

王国维在《人间词话》中说：“古今之成大事业、大学问者，必经过三种之境界：‘昨夜西风凋碧树，独上高楼，望尽天涯路’。此第一境也。‘衣带渐宽终不悔，为伊消得人憔悴。’此第二境也。‘众里寻他千百度，蓦然回首，那人却在灯火阑珊处’。此第三境也。”

本书不求成为经典，只愿能成为读者垫于脚下独上高楼的阶梯，将深度学习更高处的景色尽收眼底。

附录 A

其他深度学习工具

1. Kaldi

Kaldi 是用 C++编写的专门面向语音识别的工具集,适用于自动语音识别(Automatic Speech Recognition, ASR)系统,遵循 Apache v2.0 开源协议。

Kaldi 标志如图 A-1 所示。



图A-1 Kaldi标志

Kaldi 是基于有限状态转换机(Finite State Transducers, FST)和大量脚本构建的完整语音识别系统。核心库支持建模任意语音上下文尺寸,声学模型使用子空间高斯混合模型(Subspace Gaussian Mixture Model, SGMM)、标准高斯混合模型(GMM),以及所有常用线性和约束变换。

项目主页: <http://kaldi-asr.org/>

GitHub 地址: <https://github.com/kaldi-asr/kaldi>

2. Deeplearning4j

Deeplearning4j 是用于 Java 的深度学习框架,也是首个商用级别的深度学习开源库。Deeplearning4j 由创业公司 Skymind 于 2014 年 6 月发布,使用 Deeplearning4j 的不乏埃森哲、雪弗兰、博斯咨询和 IBM 等明星企业。DeepLearning4j 是一个面向生产环境和商业应用的高成熟度深度学习开源库,可与 Hadoop 和 Spark 集成,即插即用,方便开发者在 APP 中快速集成

深度学习功能。它可应用于以下深度学习领域：

- 人脸/图像识别
- 语音搜索
- 语音转文字（Speech to text）
- 垃圾信息过滤（异常侦测）
- 电商欺诈侦测

Deeplearning4j 标志如图 A-2 所示。



图A-2 Deeplearning4j 标志

项目主页：<http://deeplearning4j.org/>

GitHub 地址：<https://github.com/deeplearning4j/deeplearning4j>

3. Bidmach

Bidmach 是一个新的“rooflined”工具集，有大量在 CPU 或 GPU 上最快实现的机器学习算法（回归、聚类、主题模型、随机森林），深度学习层也在不断发展中。使用 Bidmach 可以快速建立单机机器学习算法原型，可以构建深度学习模型并水平扩展到集群，支持 Apache Spark。

Bidmach 标志如图 A-3 所示。



图A-3 Bidmach标志

项目主页：<http://bid2.berkeley.edu/bid-data-project/>

GitHub 地址：<https://github.com/BIDData/BIDMach>

4. Blocks

Blocks 是一个非常模块化的框架，有助于在 Theano 上建立神经网络。目前，它支持并提供的功能有：

- 构建参数化 Theano 运算，称之为 “bricks”。
- 在大型模型中使用模式匹配来选择变量以及 “bricks”。
- 使用算法优化模型。
- 训练模型的保存和恢复。
- 在训练过程中检测和分析值（训练集以及测试集）。
- 图形变换的应用，如 dropout。

GitHub 网址：<https://github.com/mila-udem/blocks>

5. Chainer

Chainer 是一个基于 Python 的开源深度学习软件框架，支持 CPU/GPU 加速。

用 Chainer 编程相对直观，代码易懂，用户可以高效地实现复杂神经网络模型。

Chainer 标志如图 A-4 所示。



图A-4 Chainer标志

项目主页：<http://chainer.org/>

GitHub 地址：<https://github.com/pfnet/chainer>

6. cuda-convnet&cuda-convnet2

cuda-convnet 和 cuda-convnet2 是深度学习大神 Alex Krizhevsky（2012 年 ILSVRC 冠军）利用 Python/C++/CUDA 编写的深度卷积神经网络框架，充分挖掘了 GPU 硬件架构，使用纹理内

存优化卷积算法达到了相当高的效率。该框架是为 GPU 定制的，因此运行这个框架要求读者必须有 GPU 硬件。

cuda-convnet 标志如图 A-5 所示。



图A-5 cuda-convnet标志

项目主页: <https://code.google.com/p/cuda-convnet/>

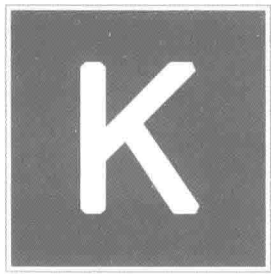
<https://code.google.com/archive/p/cuda-convnet2/>

GitHub 地址: <https://github.com/akrizhevsky/cuda-convnet2>

7. Keras

Keras 是基于 Theano 的一个深度学习框架，它的设计参考了 Torch，用 Python 语言编写，是一个高度模块化的神经网络库，支持 GPU 和 CPU。

Keras 标志如图 A-6 所示。



图A-6 Keras标志

项目主页: <http://keras.io/>

GitHub 地址: <https://github.com/fchollet/keras>

8. MatConvNet

MatConvNet 是 Matlab 上的卷积神经网络 CNN 实现。该工具包设计简单且灵活，它将 CNN 模块（用于计算滤波器组线性卷积、特征下采样等函数）封装为易用的 Matlab 函数，便于快速实现新 CNN 架构原型设计。同时，MatConvNet 支持 CPU/GPU 高效计算，能够在大规模数据

集如 ImageNet 上训练复杂模型。

MatConvNet 标志如图 A-7 所示。



图A-7 MatConvNet标志

项目主页：<http://www.vlfeat.org/matconvnet/>

GitHub 地址：<https://github.com/vlfeat/matconvnet>

9. Lasagne

Lasagne 不只是一个美味的意大利菜，也是一个与 Blocks 和 Keras 有着相似功能的深度学习库，但其在设计上与它们有些不同。Lasagne 的设计理念是：

- 简单化——易于使用和扩展的机器学习库，每添加一个特征，就应该考虑其对易用性和扩展性的影响，每一个抽象概念的加入都应该仔细检查，以确定增加的复杂性是否合理。
- 小接口——尽可能少的类和方法，尽可能依赖 Theano 的功能和数据类型，遵循 Theano 的规定，如果没有严格的要求，不要在类中封装东西，这会使它更容易使用并且扩展它。
- 不碍事——未使用的功能应该是不可见的，用户不会考虑自己不使用的功能，尽可能单独使用库文件中的组件。
- 透明性——不要试图掩盖 Theano，尽量以 Python 或 NumPy 数据类型的形式将函数和方法返回给 Theano 表达式。
- 重点——遵循 UNIX 哲学“做一件事，并把它做好”，重点集中在前馈神经网络。
- 实用主义——使普通用例更易于使用，这要比支持每一个可能的用例更为重要。

项目主页：<http://lasagne.readthedocs.io/en/latest/>

GitHub 地址：<https://github.com/Lasagne/Lasagne>

10. Marvin

Marvin 是普林斯顿大学视觉工作组新推出的 C++ 框架。该团队还提供了一个文件用于将

Caffe 模型转换成与 Marvin 兼容的模式。

GitHub 地址: <https://github.com/PrincetonVision/marvin>

11. ConvNetJS

ConvNetJS 是斯坦福大学博士生 Andrej Karpathy 基于万能的 JavaScript 开发的浏览器插件, 可以在浏览器中训练神经网络, 支持通用神经网络模块 (全连接层、非线性层)、分类 (SVM/Softmax) 和回归 (L2) 代价函数, 可描述和训练处理图片的卷积网络, 试验中的基于深度 Q 学习的增强学习模块等。Karpathy 还写了一个 ConvNetJS 的入门教程, 以及一个简洁的浏览器演示项目。

ConvNetJS 标志如图 A-8 所示。



图A-8 ConvNetJS标志

项目主页: <http://cs.stanford.edu/people/karpathy/convnetjs/index.html>

GitHub 地址: <https://github.com/karpathy/convnetjs>

深度学习是当今人工智能领域最炙手可热的技术，Caffe又是深度学习众多开源框架中很杰出的一款。永科撰写的这本著作，倾注了很多心血——既有深度学习理论知识的讲解，又有Caffe源代码的剖析，还包括解决实际问题的案例；内容翔实、思考全面、深入浅出，每章末尾还附有练习题和参考资料，是大家了解深度学习知识、实践人工智能应用的一本优秀指南。

赵永科的文章和他本人的工作态度一样诚恳。本书不仅收纳了深度学习领域的经典案例，还手把手带领读者实践了工具设置与模型搭建，并深入浅出地剖析了Caffe的底层原理，绝对是诚意十足的大作！

——简士伟 英特尔（数据中心工程事业群）平台方案架构师

本书带领你深入浅出地穿越深度学习模型，揭开它神秘的面纱；通俗易懂，实践性强，用实例引导读者从基本原理到代码实现再到应用场景，涵盖了深度学习的热门技术，是目前市面上为数不多的深度学习源码解析类参考资料，也是一本可以让你快速掌握深度学习精髓的好书！

——刘莹 中国科学院大学教授/博导，CUDA教学中心主持人/CUDA研究中心主持人

本书对深度学习的历史做了简单梳理，对深度学习的常用开源库做了非常全面的介绍，尤其是对Caffe做了非常深入的剖析，既探究了Caffe代码细节，又介绍了深度学习可视化及比赛，是一本非常实用的深度学习入门及工具书，相信本书会对国内深度学习应用的普及产生至关重要的影响。

——孙佰贵 阿里巴巴资深算法工程师

毫无疑问，深度学习是当今IT行业最火热的词汇之一。作为NVIDIA负责高性能计算团队的负责人，我看到越来越多的公司在深度学习领域大量投入。深度学习让图像识别率更高、语音识别更准确。而许多有志于在深度学习领域一展拳脚的研发人员，却苦于没有一本浅显易懂的深度学习入门书籍，来引领他们开始使用深度神经网络这一强大的工具。本书以应用广泛的Caffe为切入点，深入地介绍了Caffe的使用及一些应用实例，可以让读者对深度学习应用的开发过程有一个非常直观的理解。好比学习一门新的编程语言最有效的手段就是编写几个例子程序一样，本书正可以作为深度学习研发人员的入门快速通道。

——赖俊杰 英伟达高性能计算团队技术经理

有两个标志性的事件让深度学习进入了大众视野——谷歌大脑学会了“猫脸识别”和AlphaGo战胜了李世石。尤其后者，让人们惊呼：人工智能时代要来临了吗？是的！伴随摩尔定律下计算机运算能力的大幅提升，人工智能在越来越多领域找到有价值的落脚点。这一代的机器学习工作者无疑是非常幸运的，一个注定伟大的时代等待着大家去探索。很高兴看到国内这么快就出现了这样一本深度学习的原创书籍，而更加难能可贵的是，本书内容还是来自于作者在阿里云进行深度学习一线工作的实战总结，相信此书可以帮助大家更好地进入这个日新月异的领域。

——谷文栋 推荐技术社区ReSysChina发起人



博文视点Broadview



@博文视点Broadview

上架建议：机器学习/人工智能

ISBN 978-7-121-29115-9



9 787121 291159 >

定价：79.00元



策划编辑：张春雨
责任编辑：葛娜
封面设计：吴海燕

www.aibbt.com 让未来触手可及